TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

YUNKUN NIU

COMPARING REAL-TIME DATA ANALYTICS TECHNOLOGIES FOR REMOTE PATIENT MONITORING

Master of Science thesis

Examiner: prof. Joni Kämäräinen
Examiner and topic approved by the Faculty Council of the Faculty of Signal Processing
on 20th May 2018

ABSTRACT

YUNKUN NIU: Comparing Real-Time Data Analytics Technologies for Remote Patient Monitoring
Tampere University of technology
Master of Science Thesis, 45 pages, 0 Appendix pages
May 2018
Master's Degree Computer Science in Information Technology
Major: Data engineering
Examiner: Professor Joni Kämäräinen

Keywords: remote patient monitoring, real-time, data analytics, Apache Storm, Apache Spark Streaming, Apache Kafka Steams, stream processing, distributed system.

With the maturing of Big Data and telecommunication technologies, it has become possible to implement remote patient monitoring services for remote diagnosis. These services allow patients to be monitored regardless of their physical location. The more efficient these services are, the better the patients will be taken care of. Patients feel more secure with real-time monitoring because they know that they will receive an instant diagnosis when something anomalous happens in their bodies.

The ease of developing and maintaining real-time data analytics technologies for remote patient monitoring services is a central factor in the development of real-time monitoring systems. An easy technical solution can help R&D teams to continuously deliver new versions of services to patients. Hence, patients can benefit from regularly updated service versions compared with traditional location-bound healthcare services. More complex technologies always requires more learning time and attention from developers. Therefore, selecting an easy programming technology can have a significant impact on providing real-time remote patient monitoring services.

This thesis introduces three stream processing technologies that are popular both in the industry and academia. They all come from the open source Apache Foundation: Storm, Spark Streaming and Kafka Streams. The thesis first introduces the architecture and core concepts of the three technologies at high level. Then the author designs an experimental environment to compare the ease of programming and performance. Finally, the author studies the design philosophies of the three technologies and gives a detailed comparison of the internal implementations of the key features.

ACKNOWLEDGEMENTS

I would like to especially give my gratitude to my supervisor Antero Taivalsaari for his sustained help during the entire long-term process of my master thesis. Especially in the beginning when I had little clue on how to plan the thesis project, Antero outlined the structure of the thesis. That boosted the thesis project as I would have a clear goal and plans to proceed to the goal step by step. More than this, Antero also met with me regularly to give instant feedback to my work and answer my questions patiently. I deeply feel the frequent meetings cost plenty of his time and attention. Thanks Antero for so much of help.

I also would like to appreciate all my colleagues at Nokia Technologies, especially Petri Selonen and Juha Uola. Petri gave me the thesis worker position, and he also helped me get familiar with everything at Nokia Technologies. Juha gave me so much help and knowledge on Amazon Web Services. Without him, I would definitely had needed more time to finish this project. And so are my other colleagues who took good care of me. I had a quite pleasant time at Nokia Technologies.

I give my greatest thanks to Joni Kämäräinen for his serious review of my thesis. When I first saw his hand written comments on my thesis, that really touched me.

Last but not least, I want to say thank you to my parents and my friends who patiently talked with me when I lost focus or felt upset. Master Thesis writing is a long-term project, and they give me so much encouragement along the way.

Tampere, 20.05.2018

Yunkun Niu

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

ECG	Electrocardiography
RPM	Remote Patient Monitoring
TUT	Tampere University of Technology
JSON	JavaScript Object Notation
UDF	User Define Function
API	Application Programming Interface
YARN	Yet Another Resource Negotiator
AWS	Amazon Web Services
CPU	Central Processing Unit
RAM	Random Access Memory
GB	Gigabyte
DAG	Directed Acyclic Graph
R&D	Research and Development
IO	Input and Output
SQL	Structured Query Language
BSD	Berkeley Software Distribution
HDFS	Hadoop Distributed File System
RDD	Resilient Distributed Datasets

1. INTRODUCTION

New technologies are being created continuously to make human lives better. At this moment, the majority of patients still have to stay in hospitals under professional monitoring in case of accidents. No doubt this is necessary in the highly risky stage such as the first few days after surgery. However, once the risk level is lower, patients would likely prefer to get back to their normal lives as soon as possible. Their familiar home, families and friends are the best cure for them. But there is also a high rate of recurrence and death in the first few weeks after patients leave hospitals to home because of irregular schedules in taking medicine, lower hygiene level, the unstable emotions and so many other reasons. If new technologies can remotely monitor the patients' health status in real-time, then their health should be guaranteed almost as well as when they are staying in a hospital. And more than that, they can enjoy so many things that hospitals typically do not have, such as private space, favourite food and funny TV programs. The author of the thesis aims to find out the technologies that help patients have better recovering time.

A human produces activity signals such as heartbeat, blood pressure, temperature and so on until the signs of life are gone. Doctors and clinicians can diagnose patients' illness by analysing the features of these activity signals. Medical instruments have been able to measure these signals to support diagnosis for a long time. Patients do not feel too uncomfortable if they are requested to do some examinations with these instruments. And there has been decades of research and development on home care services. Patients would feel more comfortable if they can stay in their familiar home environment and at the same time feel secure with the real-time remote patient monitoring services.

Every second, a large volume of health data is being generated by patients. This brings a big technical challenge to real-time remote patient monitoring services. Because typically a remote patient monitoring service is required to serve millions of people. The recorded data are disordered, unreadable and meaningless for clinicians if they are not processed. Thus, data analytics technologies are required to be able to finish processing large amounts of data within a short time period. This is the technical challenge for real-time RPM services. The less time the analysis takes, the more time clinicians will gain for diagnosis, so patients can feel more secure.

With the fast development of Big Data technologies during the past decade, there is a variety of options available in the open source community and industry. A set of common features supported by them are high throughput, high availability, low latency, distributed processing and scalability. Definitely, all these features are beneficial for a system's stability and efficiency in the area of remote patient monitoring. But the ease of programming and maintenance are also important factors in applying a new technology, as it affects the R&D progress and quality of the service provider. Furthermore, they affect the service cycle and quality for patients considerably.

This thesis selects a few mainstream real-time analytics technologies for remote patient monitoring and compares their advantages and disadvantages. Based on the popularity in both academia and industry, the author of the thesis selected three open source technologies from Apache Software Foundation: Storm, Spark Streaming and Kafka Streams. During the learning process, the author found that their design philosophies are unique and worthy of a deeper study. Thus this thesis focuses on comparing their architecture, ease of programming and design philosophies.

The main topic of this thesis is the comparison of real-time data analytics technologies for remote patient monitoring. Chapter 2 'background' introduces the key concepts that are involved in this topic. The author aims to give readers the required background knowledge before getting into the next detailed technical chapters. Chapter 3 'Comparing the architecture' introduces the fundamental concepts, architecture and working mechanism. The author believes that every engineer has to understand these aspects if they want to use the selected technologies well. Next, Chapter 4 compares the ease of programming by describing an experimental environment in which the author defines a common abstracted program flow and implements it with the three technologies to compare them in terms of the ease of programming. Chapter 5 'comparing the design philosophies' is the author's primary area in the whole thesis. It discusses the parallelism mechanisms, scalability mechanisms, fault tolerance mechanisms and the guaranteed message processing mechanisms that should be the minimum required four features that any real-time data analytics technologies should have. Chapter 6 'conclusion' is the author's reflection on the knowledge that was gained during the whole process of comparing real-time data analytics technologies for remote patient monitoring.

2. BACKGROUND

2.1 Remote Patient Monitoring

Remote patient monitoring (RPM), a.k.a., telemonitoring, involves the passive collection of physiological and contextual data of patients in their own environment using medical devices, software, and optionally environment sensors. The collected data is transmitted to the remote care provider, either in real-time or intermittently, for review and intervention.[1]

The benefits of RPM can be grouped into economic benefits for the financial risk holders, management benefits for the healthcare providers, and quality of life benefits for the patients. A number of studies have already shown dramatic reductions in key cost drivers for the healthcare community through RPM technology. RPM also supports effective and efficient population management through automated monitoring to prevent hospitalisations. Last, patients can feel more secure in terms of quality of life with this type of supervision.[2]

This thesis focuses on the technologies that are used for analysing the patients' data gathered by the sensors to support remote monitoring and diagnoses in real-time.

2.2 Data Analytics

The term *data analytics* became popular in the early 2000s[3,4]. Data analytics is defined as the *application of computer systems to the analysis of large datasets for the support of decisions*[5]. The data analysis is the process of extracting valuable information from the raw data. The resulting information is used to support decision making or recommend actions.

There are many reasons why data analytics is important in the medical context. First, the inflow of health data can be both voluminous and too detailed for humans to process without automated data analysis tools. Second, simply performing periodic reviews of the data can add significant latency between the occurrence of a medically significant event and the (possibly necessary) reaction by a caregiver. Third, manual analysis can miss relevant subtleties in the data, particularly when an actionable event is the result of a combination of factors spread across several data streams.[6] Therefore this thesis compares real-time data analytics technologies in terms of large data processing and ease of use for RPM.

Typically, data analytics is a combination of the following steps: loading, transformation, validation, sorting, summarization and aggregation and visualisation. The visualisation part is not included in this thesis. The thesis compares the ease of programming and internal designs among the mainstream technologies in the area of data analytics.

2.3 Real-Time System

A *real-time system* is a computer system that must satisfy bounded response time constraints or risks severe consequences, including failure[7]. The response time is the time between the presentation of a set of inputs to a system and the realisation of the required behaviour. The response time is also called *latency* nowadays.

Real-time systems are characterised by computational activities with stringent timing constraints that must be met in order to achieve the desired behaviour. A typical timing constraint on a task is the deadline, which represents the time before which a process should complete its execution. Depending on the consequences of a missed deadline, real-time tasks are usually distinguished in three categories:[8]

- **Hard real-time**: A real-time task is said to be hard if missing its deadline may cause catastrophic consequences on the system under control.
- **Firm real-time**: A real-time task is said to be firm if missing its deadline does not cause any damage to the system, but the output has no value.
- **Soft real-time**: A real-time task is said to be soft if missing its deadline has still some utility for the system, although causing a performance degradation.

In this thesis, the actual meaning of the term real-time is soft real-time. This is because in the RPM area that this thesis focuses on a missed deadline will not cause critical damages, but it could have influences on the diagnosis and patients' experience.

Real-time data analytics is different from offline data analytics. The input data set of an offline data analytics system has a few aspects, fixed, bounded and high volume. There are no deadline requirements for the output of an offline data analytics system. While for a real-time data analytics system, the time requirements are critical. It must process the input data and output the results within a small time period which is the latency. With lower latency, the system can detect the changes of patients' status quicker after the changes happen, and then doctors can get faster into diagnosis. Patients will thus have more chances to recover when something anomalous occurs. Thus this thesis focuses primarily on real-time data analytics technologies that can process a large amount of data with very low latency.

2.4 Data Stream and Stream Processing

The input data of a real-time data analytics system has three unique characteristics that sets it apart from other types of systems. These characteristics are: 1) always on always flowing, 2) loosely structured and 3) high-cardinality storage. Always on means that the data is always available and new data is always being generated. Streaming data is often loosely structured compared to many other datasets. Part of the reason seems to be that Streaming data comes from a variety of sources. Cardinality refers to the number of unique values a piece of data can take on.[9] The input data keeps flowing like a stream.

Thus a data stream in this thesis represents a continuous unbounded *data stream* as the input of a system.

Stream Processing is a computer programming paradigm. In the context of the data stream, stream processing in this thesis specifies the programming patterns which process data streams as the input, generating output results continuously. Because of the three aspects of the data stream, stream processing architectures are usually high-available, low-latency and horizontally scalable.

In the RPM domain, patients generate biological data all the time. These data streams continuously flow into real-time analytics systems and wait to be processed and responded in a short time period. Stream processing programming paradigm perfectly matches this RPM scenario, and it is also the architecture of the real-time analytics technologies selected in this thesis.

2.5 Parallel Computing & Distributed System

Parallel computing is a type of computation in which many calculations or execution of processes are carried out simultaneously[10]. Compared with Serial Computing, which sequentially executes a discrete series of instructions of a problem one after another on one processor, parallel computing can simultaneously execute those instructions on different processors. Thus, parallel computing has advantages in saving time and solving larger complex problems. This is important because many problems are too large for a single computer to solve.

A *distributed system* is a collection of independent computers that appear to its users as a single coherent system[11]. It is also called a *cluster* in this thesis. In modern computer programming technologies, multi-threading and multi-processing are the main programming methods to implement parallel computing on a single computer. A distributed system can dispatch the computing tasks on several computers in the cluster to further improve the parallelism and system capacity. Thus it is usually used for large computing tasks that beyond the capacity of a single computer and time critical tasks. A distributed system provides the horizontal scalability to dynamically adjust the size of the cluster to the size of computing tasks.

For real-time analytics technologies in the RPM environment, the system can collect a huge amount of data in every second. A distributed system can improve the parallelism for computing the input data streams and thus shorten the latency. Furthermore, once a computer in a cluster crashes down, the cluster can automatically migrate its tasks to another healthy computer. This improves the high availability for real-time analytics technologies.

3. COMPARING THE ARCHITECTURE

This chapter introduces the real-time data analytics technologies that were chosen for comparison: *Apache Storm*, *Spark Streaming* and *Kafka Streams*. The author selected these technologies because of their popularity and unique representative design. All of them have been widely applied both in the industry and in the academia. Their names have appeared frequently in Google searches and in people's discussions when searching for technologies for real-time analytics and stream processing.

The chosen technologies are created in the background of Apache Hadoop that is the representative of Big Data technology and batch processing technology. Hadoop is an open source framework that allows distributed processing of large data sets across clusters of computers using the MapReduce programming model with which users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key[12]. However, Hadoop's batch processing model and high file I/O make it too slow to meet the requirements for real-time analytics. This is the primary challenge that the selected three technologies are addressing. Apache Storm was created as the 'Hadoop of real-time data analytics'. Almost at the same time, Apache Spark was created with the Spark Streaming library for stream processing. A few years later, Kafka Streams was created to make stream processing easies.

Although the selected three technologies all focus on the same problem, their designs and internal implementations are quite unique and representative. This chapter focuses mainly on technology introductions. In the next chapter, the thesis will then compare the internal design thinking among them.

3.1 Storm

3.1.1 Introduction

Apache Storm is officially defined as a *free and open source distributed real-time computation system*[13]. Apache Storm is designed for reliable processing unbounded streams of data in real-time.

Apache Storm was originally created by Nathan Marz while he was in the company Backtype. Backtype was acquired by Twitter, and then later on September 19th, 2011 Twitter open sourced Storm to GitHub. After that Storm project entered the Apache Incubator project status on September 18, 2013, and graduated as a top-level Apache project on September 17, 2014. Apache Storm has been used widely for real-time analytics in Internet companies such as Yahoo, Twitter, Spotify and so on.

Storm has several essential features that make it trusted and user-friendly for real-time data processing. Storm provides three levels of strategies to guarantee that input data will be processed with best effort at least once or exactly once. It also supports multi-language programming so that developers with different backgrounds can get started with it easily. Especially, Storm has already been integrated with existing message queue and database technologies such as Kafka and HBase. A Storm application can consume streams of data from them and then produce new data back to them.

3.1.2 Key Concepts

Stream

As introduced above Storm processes unbounded streams of data. The stream is an abstracted concept in Storm. A stream is an unbounded sequence of tuples. The tuple is a data structure of Storm and a tuple stores a list of values that are the real data to be processed. Storm provides the primitives to transform one or multiple streams to one or multi-streams. The Figure 3.1 shows the logical concept of a stream. The circles inside of it represent tuples.

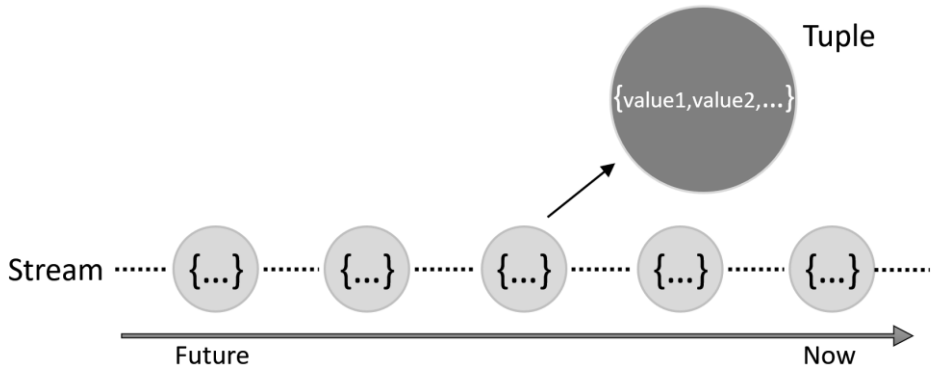

Figure 3.1 Stream and Tuples

Topology

Topology is the graph of logic for a Storm application. In general, topologies are the applications people create in Storm. Figure 3.2 illustrates the structure of a Storm topology. The graph of a topology consists of a layer of Spouts and one or multiple layers of Bolts that are connected with streams. The Spout and Bolt are two core concepts in Storm as well.

A Spout is the source of streams as a start of a topology. The Spouts read data from external data resources (e.g., database and message queue) then emit them to the Bolts in the topology. Spouts can emit more than one streams to different Bolts to process respectively.

Bolts are the real processing nodes in a topology. Bolts can read multi-streams and emit multiple new streams. Basically, Bolts can do all types of transformations on the data inside of the Bolts.

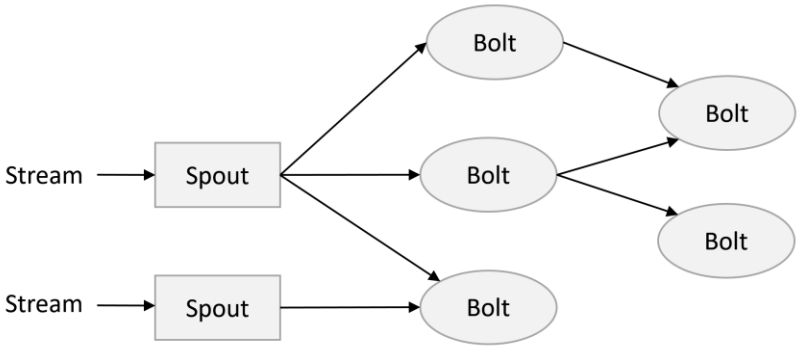

Figure 3.2 Topology graph of Storm

3.1.3 Architecture

The topologies run in a Storm cluster. Since Storm is a scalable distributed fault-tolerant computation system, Storm consists of several core components to guarantee topologies to successfully and stably run inside of it. Storm cluster has two types of nodes: the master nodes and worker nodes. Figure 3.3 shows the architecture of a Storm cluster.

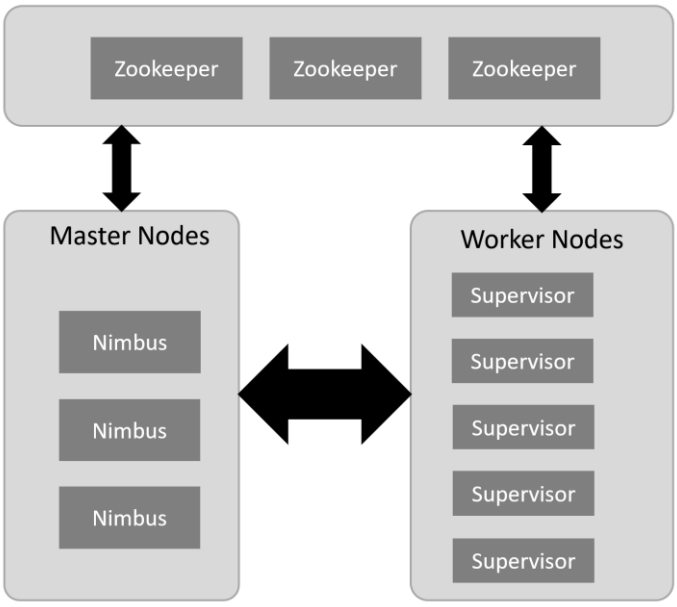

Figure 3.3 Storm cluster architecture

One Storm cluster can have multiple master nodes. Each master node has a daemon process named "Nimbus". Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures.

Each worker node has a daemon process named "Supervisor". The supervisor listens for assigned work to its machine and starts and stops worker processes as necessary based on

what Nimbus has assigned to it. Each worker process executes a subset of a topology; a running topology consists of many worker processes spread across many machines.

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services[14]. Here, it provides coordination between Nimbus and Workers. The Nimbus daemon and Supervisor daemons are fail-fast and stateless; all state is kept in Zookeeper or on local disk. If any processes of master nodes and workers nodes crash, Nimbus and Supervisors can recover them from the crash point. This guarantees the stability of the system.

Storm UI is a web tool for developers to monitor the states of topologies and the Storm cluster. It summaries the key parameters of a cluster, Nimbus daemons, Supervisor daemons and topologies stats.

3.2 Spark Streaming

3.2.1 Introduction

Spark Streaming makes it easy to build scalable, high-throughput, fault-tolerant stream processing applications[15]. Spark Streaming is not an independent technology but an extensive library of the core Apache Spark API. Spark is a fast and general engine for large-scale data processing. Beside Spark Streaming, Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning and GraphX. Figure 3.4 demonstrates the relationships among them.

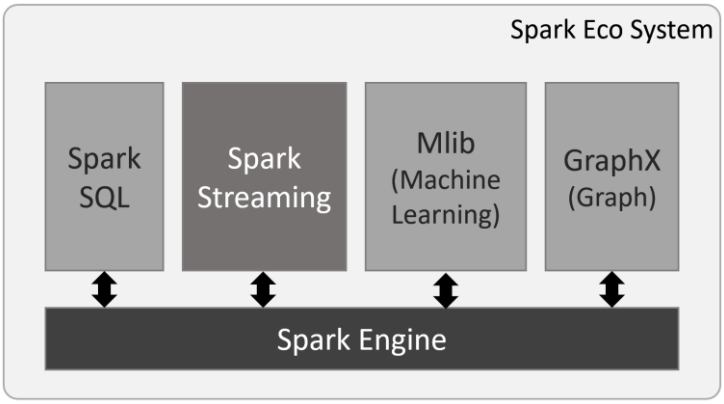

Figure 3.4 Spark eco system

Spark was initially started by Matei Zaharia at UC Berkeley's AMPLab in 2009, and open sourced in 2010 under a BSD license. On June 19th, 2013, the project entered the incubation of Apache Software Foundation and switched its license to Apache 2.0. On February 19th, 2014, Spark graduated as a top-level Apache Project.

Spark Streaming has several highlights that make it one of the most popular open source large-scale seamless data stream processing technologies. First, Spark Streaming offers a set of high-level operators for developers to easily build data stream processing

applications. Second, to broaden its user range, it supports three mainstream programming languages: Scala, Java and Python. Third, Spark Streaming is fast because it uses Spark as its computation engine. Spark officially claims it runs programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk[16]. Last, by running on Spark, Spark Streaming can reuse the same code for batch processing, join streams against historical data, or run ad hoc queries on stream state, and thus build powerful interactive applications and not just analytics. Spark Streaming is also fault tolerant. Spark Streaming recovers both lost work and operator state (e.g., sliding windows) out of the box, without any extra code on the programmers' part.

3.2.2 Data Processing Mechanism

Logically, Spark Streaming receives live input data streams and divides the data into micro batches, which are then processed by the Spark engine to generate the final stream of results in batches. Figure 3.5 illustrates the data flow.

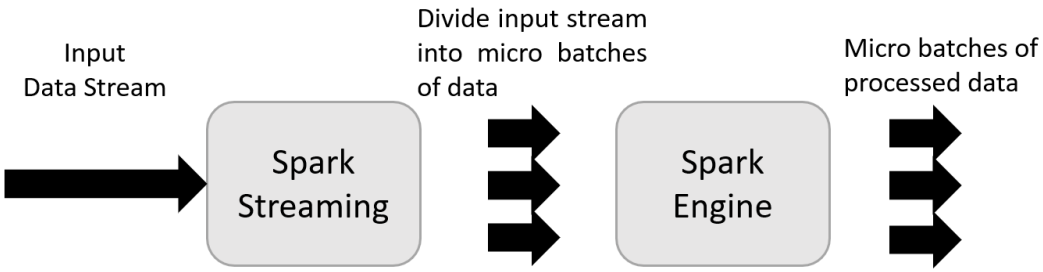

Figure 3.5 Spark Streaming data flow

3.2.3 Key Concepts

Resilient Distributed Datasets (RDDs)

Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of read-only collection of elements partitioned across the distributed computer nodes in memory which can be operated on in parallel. To achieve fault-tolerant property, there are two ways to create RDDs: parallelizing an existing collection in the driver program or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format. An RDD could keep all information about how it was derived from other datasets to compute its partitions from data in table storage. Therefore, RDDs are fault tolerant because they could be reconstructed from a failure with this kept information.

RDD supports two different kinds of operations: transformation and action. When a transformation operation is called on an RDD object, a new RDD returned and the original RDD remains the same. For example, the map is a transformation that passes each element in RDD through a function and returns a new RDD representing the results. Some

of the transformation operations are Map, Filter, FlatMap, GroupByKey, and Reduce-ByKey.

An action returns a value to the driver program after running a computation on the RDD. One representative action is reduce that aggregates all the elements of the RDD using some function and returns the final result to the driver program. Other actions include collect, count, and save.

Discretized Streams (DStreams)

Internally, Spark Streaming provides a high-level abstracted data structure called DStream (discretized stream), which represents a continuous stream of data. Internally, a DStream is consist of a continuous serious of RDDs.

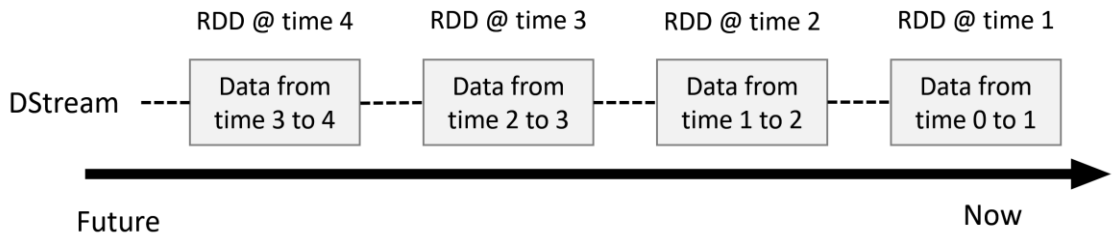

Figure 3.6 DStream and RDDs

The DStream integrates the high-level data operation functions such as map, reduce, join and window. DStreams can be created either from input data streams from sources such as Kafka, Flume, Kinesis and TCP sockets, or by applying high-level operations on other DStreams.

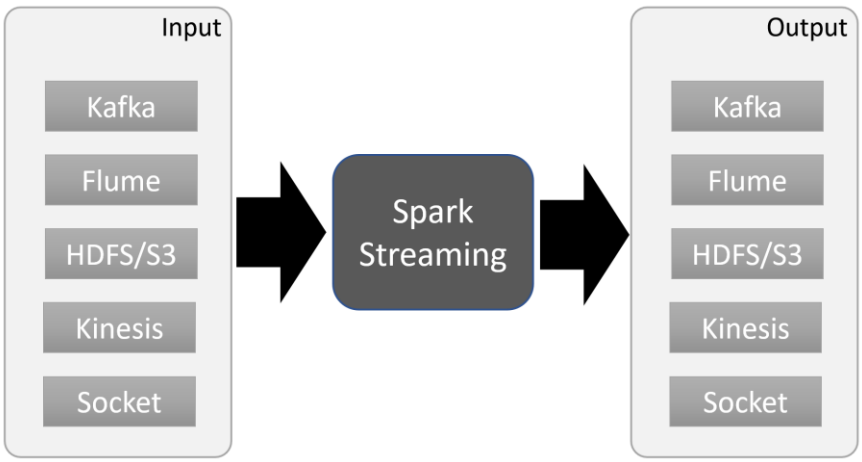

Figure 3.7 The input and output of DStream

3.2.4 Architecture

Spark Streaming applications run in a Spark cluster. Since Spark is a scalable distributed fault-tolerant computation engine, Spark consists of several core components to guarantee all Spark applications to successfully and stably run inside of it. Spark has three core

components: SparkContext, Cluster Manager and Worker Node. Figure 3.8 shows the architecture of a Spark cluster.

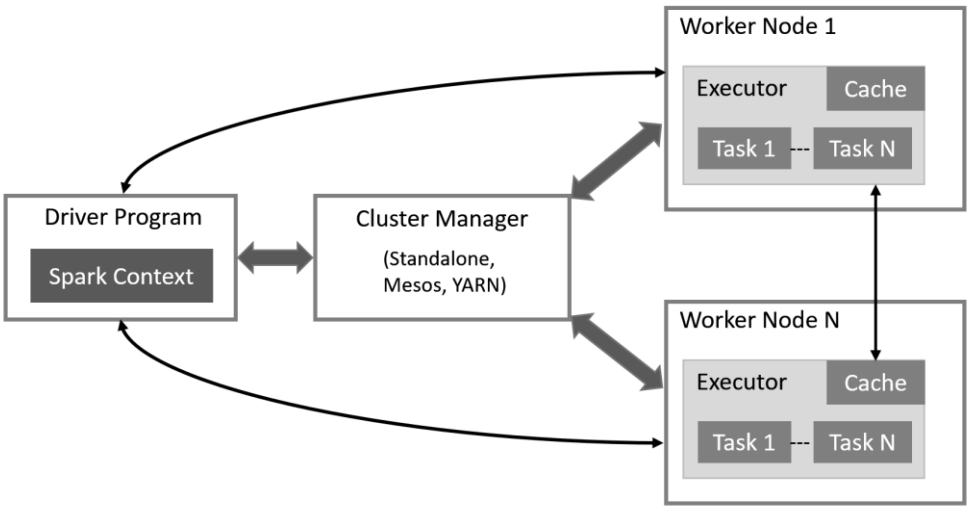

Figure 3.8 Spark cluster architecture

SparkContext is a Spark object in a Spark application program. It is also called Driver Program. SparkContext coordinates all the Spark applications to run as independent sets of processes in a cluster. Its work starts from connecting to Cluster Manager and then acquiring executors on nodes in the cluster, which are processes that run computations and store data for Spark applications. Next, it sends Spark application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends tasks to the executors to run.

Cluster Manager is responsible for managing CPU, memory, storage, and other computer resources in a cluster. Spark cluster supports three types of Cluster Managers: Standalone, Apache Mesos and Hadoop YARN. Standalone is offered by Spark by default. It is the easiest option to set up a spark cluster. Apache Mesos is a general resource manager enabling fault-tolerant and elastic distributed systems to easily be built and run effectively. Spark naturally compacts with Mesos because Spark was initially created on top of Mesos when it was built. Hadoop YARN is a framework for job scheduling and cluster resource management in Hadoop 2.0.

Spark also offers a Web UI tool for developers to monitor the stats about running tasks, executors, and storage usage. It is launched by SparkContext, listening on port 4040 by default.

3.3 Kafka Streams

3.3.1 Introduction

Kafka Streams is a client library for processing and analyzing data stored in Kafka and either write the resulting data back to Kafka or send the final output to an external

system[17]. It builds upon important stream processing concepts such as properly distinguishing between event time and processing time, windowing support, and simple yet efficient management of application state.

Kafka is an open source, high-throughput, low-latency, distributed message system. It gets used for two broad classes of applications: building real-time data pipelines for transferring data between systems and applications, and building real-time streaming applications that transform or react to the streams of data.

Kafka was originally created inside of LinkedIn as a central data pipeline. On July 4th, 2011 LinkedIn open sourced it to Apache Software Foundation. Later, after 15 months it graduated as a top-level project on October 23rd, 2012. Kafka Streams was released together with Kafka 0.10 release on March 10th, 2016.

Kafka is running in production use in thousands of companies. There are two main advanced techniques that make it so popular. First, Kafka's partitioned and replicated storage structure lets it safely store massive data in a distributed and scalable fashion to achieve high-throughput and fault-tolerant goals. Second, its simple pub/sub queue data structure and user model make it extremely low latency for building real-time applications and relatively easy for developers to learn it. Therefore, Apache Kafka is widely used for messaging, tracking web activity, monitoring, log aggregation, stream processing, event sourcing and committing log.

However, Apache Kafka only provides producer and consumer APIs for third party frameworks to process data in real-time. Kafka itself cannot transform data but focuses on data acquisition and latency-free storage. Before Kafka Streams, some real-time technologies such as Apache Storm, Spark Streaming Flink and Amazon Kinesis, took the positions of real-time analytics. But they are either near real-time or too complex. So the Kafka team introduced *Kafka Streams* to make stream processing simple. Kafka Streams is just a single lightweight Java client library without any dependencies. Real-time analytics applications can easily include it and deploy in any way, locally or in distribution. It offers a high-level Streams DSL (Domain Specific Language)[18] for easy programming. It also employs one-record-at-a-time processing to achieve millisecond processing latency for real-time analytics.

3.3.2 Key Concepts

Stream

A stream is an abstraction that represents an ordered, replayable, and fault-tolerant sequence of immutable data records, where a data record is defined as a key-value pair[19].

Stream Processing Application

A stream processing application is any Java Program that uses the Kafka Streams library to define the computational logic through one or more processor topologies.

Processor Topology

The processor topology is a logical abstraction for stream processing codes. It is a logic graph that defines the directions of data streams and where are the streams processed. Figure 3.9 shows a processor topology that processes data streams from up to down. The processor topology is made of several nodes that are connected by edges. The node is called *Stream Processor*. Each node represents a processing step, e.g., transforming data and merging data streams. The edges represent the directions that the data flow.

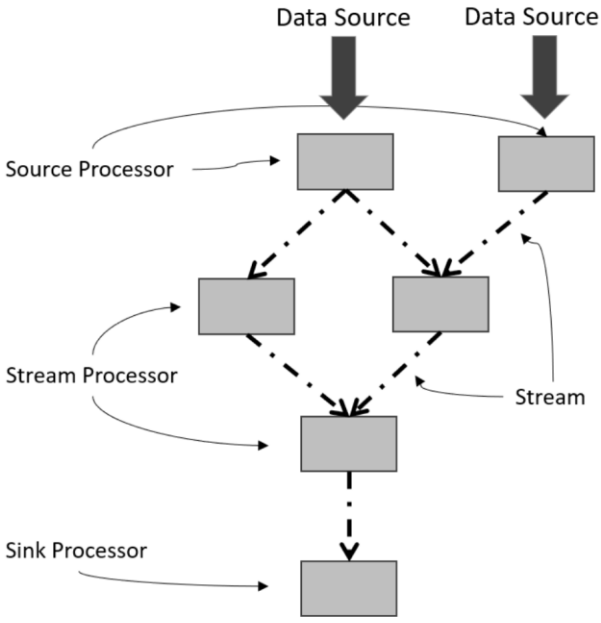

Figure 3.9 Processor topology graph

Time

In a real-time stream processing technology time is a critical aspect. There are three notions of time in Kafka.

- **Event Time**: The point in time when an event or data record occurred, i.e. was originally created “by the source”.
- **Processing Time**: The point in time when the event or data record happens to be processed by the stream processing application.
- **Ingestion Time**: The point in time when an event or data record is stored in a topic partition by a Kafka broker.

From Kafka 0.10.x onwards, timestamps are automatically embedded into Kafka messages.

3.3.3 Architecture

Architecture of Kafka

Kafka Streams is an independent Java library for stream processing of data inside Kafka. Kafka Streams provides the simplest APIs to fetch data from Kafka as a consumer and write the results data back to Kafka as a producer or send the final output to an external system. Its simplicity is based on the underlying Kafka system that guarantees and performs most work in areas such as distribution, scalability, high-throughput and fault-tolerance. Figure 3.10 shows the architecture of Kafka.

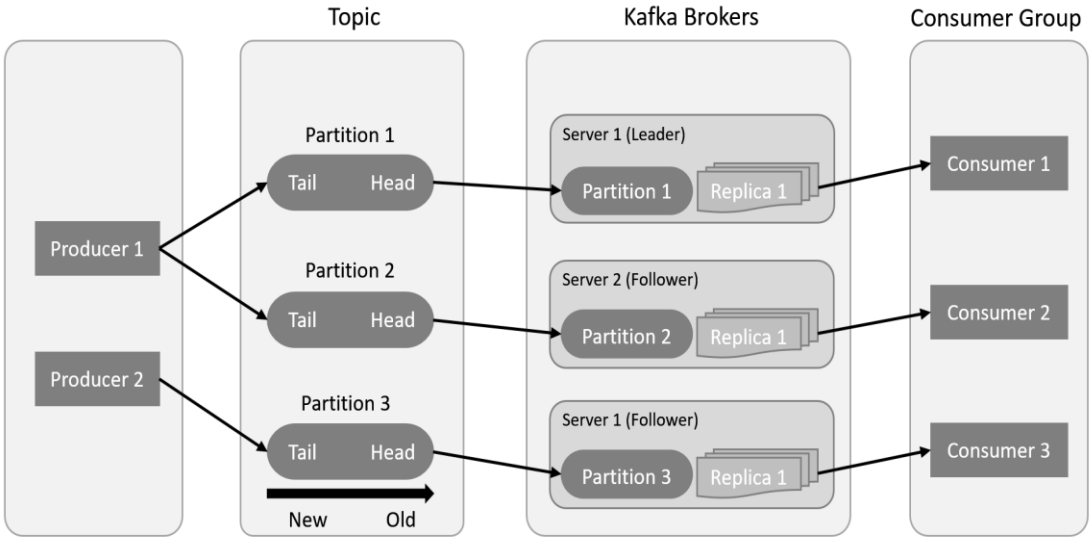

Figure 3.10 Kafka Architecture

Inside Kafka, a stream of data belonging to a particular category is called a topic. Physically, a topic is split into partitions, and the partitions are randomly stored in different servers for distribution and scalability. Every time a producer publishes a message to a broker, the broker simply appends the message to the last segment file of a partition. The producer can also choose a specific partition to send messages. Brokers are a simple system responsible for maintaining the published data. Each broker may have zero or more partitions per topic. A topic can have one or more replications. Each replication will be stored in different servers to guarantee the safety of data.

Stream Partitions and Tasks

Kafka Streams simplifies application development by building on the Kafka producer and consumer libraries and leveraging the native capabilities of Kafka to offer data parallelism, distributed coordination, fault tolerance, and operational simplicity. There are close links between Kafka Streams and Kafka in the context of parallelism. Kafka Streams uses the concepts of partitions and tasks as logical units of its parallelism model. Each stream partition is a totally ordered sequence of data records and maps to a Kafka topic partition. A data record in the stream maps to a Kafka message from that topic. The keys of data records determine the partitioning of data in both Kafka and Kafka Streams.

An application's processor topology is scaled by breaking it into multiple tasks. More specifically, Kafka Streams creates a fixed number of tasks based on the input stream

partitions for the application, with each task assigned a list of partitions from the input streams.

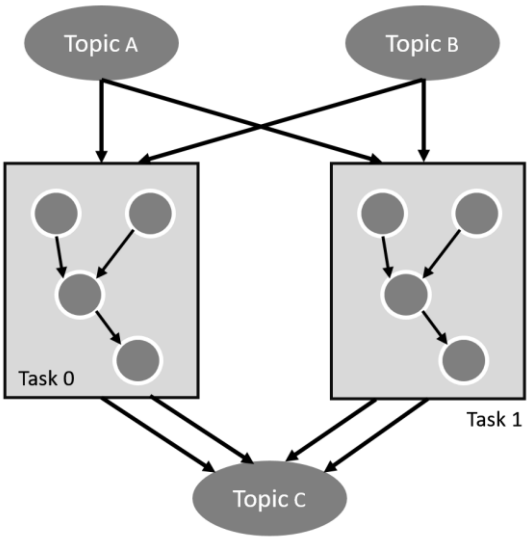

Figure 3.11 Processor topology and tasks

Parallelism

Kafka Streams allows the user to configure the number of threads that the library can use to parallelize processing within an application instance. Each thread can execute one or more tasks with their processor topologies independently.

Scaling a stream processing application with Kafka Streams is easy: developers merely need to start additional instances of the application, and Kafka Streams takes care of distributing partitions amongst tasks that run in the application instances. It is possible to start as many threads of the application as there are input Kafka topic partitions so that, across all running instances of an application, every thread (or rather, the tasks it runs) has at least one input partition to process.

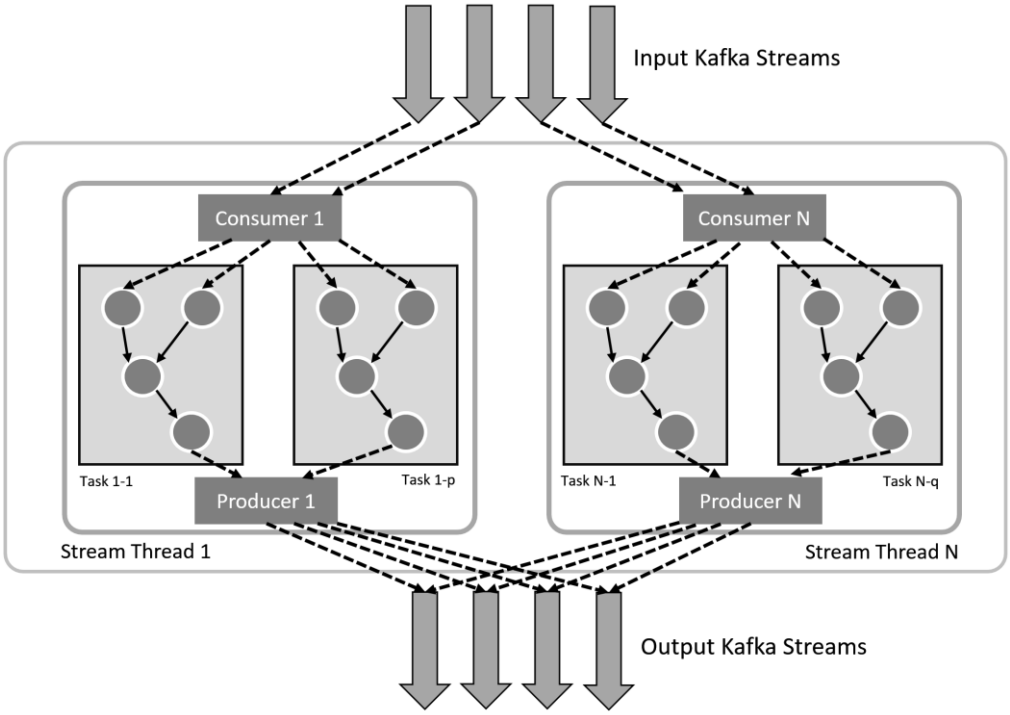

Figure 3.12 The parallelism mechanism of Kafka Streams

3.4 Other Real-Time Analytics Systems

Apache Storm, Spark Streaming and Kafka Streams are good representative systems for real-time data analytics. The author has chosen them mainly because of their popularity and the original thesis work assignment. In this field, there are also other technologies such as Apache Flink and Apache Samza that would be worthy of studies. To avoid losing focus, however, this thesis does not go into detailed research on them. But the thesis still introduces them briefly below so that readers can have an idea of the alternatives.

3.4.1 Flink

Apache Flink is a free and open source stream processing framework for distributed, high-throughput, high-available, and accurate data streaming applications[20]. Flink supports two types of datasets: the unbounded ones (infinite datasets that are appended to continuously) and the bounded ones (finite un-changing datasets). Thus, Flink also has two execution models respectively: the streaming model (continuously executing as long as data is being produced) and the batch mode (executing until completion in a finite amount of time).[21]

The original name of Flink is Stratosphere. Stratosphere was a research project whose goal was to develop the next generation Big Data Analytics platform. The project included universities from Berlin area, namely TU Berlin, Humboldt University and the Hasso Plattner Institute.[22] Later on April 14th, 2011 the system was open sourced and entered the incubation phase in Apache Software Foundation. On December 17th, 2014 Flink graduated as a top-level project.

Flink has three features that make it outstanding for real-time data analytics. First it provides results that are accurate, even in the case of out-of-order or late-arriving data. Second it is stateful and fault-tolerant and can seamlessly recover from failures while maintaining exactly-once application state. Last, it performs at large scale, running on thousands of nodes with very good throughput and latency characteristics.[21]

3.4.2 Samza

Apache Samza is a distributed stream processing framework. It uses Apache Kafka for messaging, and Apache Hadoop YARN to provide fault tolerance, processor isolation, security, and resource management.

Same as Kafka, Samza was originally invented inside of LinkedIn who is the world's largest professional network with more than 562 million users in more than 200 countries and territories worldwid[23]. The existing ecosystem at LinkedIn had a huge influence on the motivation and architecture of Samza. Because LinkedIn had integrated Kafka with almost all the data, there was a need to do a lot of stream processing. Samza was designed to read messages from Kafka and do some processing, and then write messages back. On July 30th, 2013 Samza entered the incubation phase in Apache Software Foundation and then graduated as one of the top-level projects on January 2nd, 2014.

4. COMPARING THE EASE OF PROGRAMMING

The ease of programming means developers' feelings about how easy it is to code with a technology. The comparison result can be very subjective because it depends on programmers technical backgrounds and preferences. The thesis does not provide numerical data to give an objective comparison. Instead, the author sets up an experiment in which the author defines a programming target and implements the target with the three technologies. By this experiment, readers can have their own subjective feelings about the ease of programming with each technologies.

4.1 Experiment

To compare the ease of programming of the three selected real-time analytics technologies, a common experiment environment needs to be created for testing them. The environment must be fair to each technology. The environment consists of a common data source, a group of computers with a common configuration, and a program flow to be implemented with each technology.

This chapter first illustrates the big picture of the architecture of the experiment. It shows the relationships among all the services in this experimental environment. Next, the chapter introduces the program flow. It is a logical model and all the stream processing applications follow its instructions and execute the model step by step. A common data generator is also required for feeding the same data to all the three technologies. The data generator is literally the data source in the experiment, generating data for stream applications to process. In the end, statistics are introduced for comparing the performance of each technology.

To make the research result more approachable to the existing system of Nokia, an internal tool called Signal Generator was used as data generator in the experiment. Signal Generator can simulate human hearts' ECG data and send the data to Kafka Pipeline.

4.2 Architecture of the Experiment

Storm, Spark Streaming and Kafka Streams share some common external dependencies: Kafka as the data pipeline and the direct data resource, and Zookeeper as the node coordinator and Data Generator. To provide a fair experiment environment, all these services are deployed in the same cluster for common usage. Besides, one powerful computer is selected as the common executor for every technology. All the computing tasks of each technologies will be deployed and executed on this executor.

As shown in Figure 4.1, the frame on top represents the cluster onto which Kafka, Zookeeper and Data Generator are deployed. The Data Generator generates simulated data to the partitions of a Kafka topic. Kafka just saves the data locally and waits for the

stream processing applications of the three technologies to consume the data. There are two main reasons to apply Kafka as the data pipeline in this architecture. The first, it is very common to apply Kafka in near real-time data processing architectures. Kafka runs in production in thousands of companies. The second, Kafka is already the direct data source for real-time analytics technologies in Nokia's remote patient monitoring architecture. To let the research results be compatible with the existing architecture, the thesis deploys the exactly same version of Kafka.

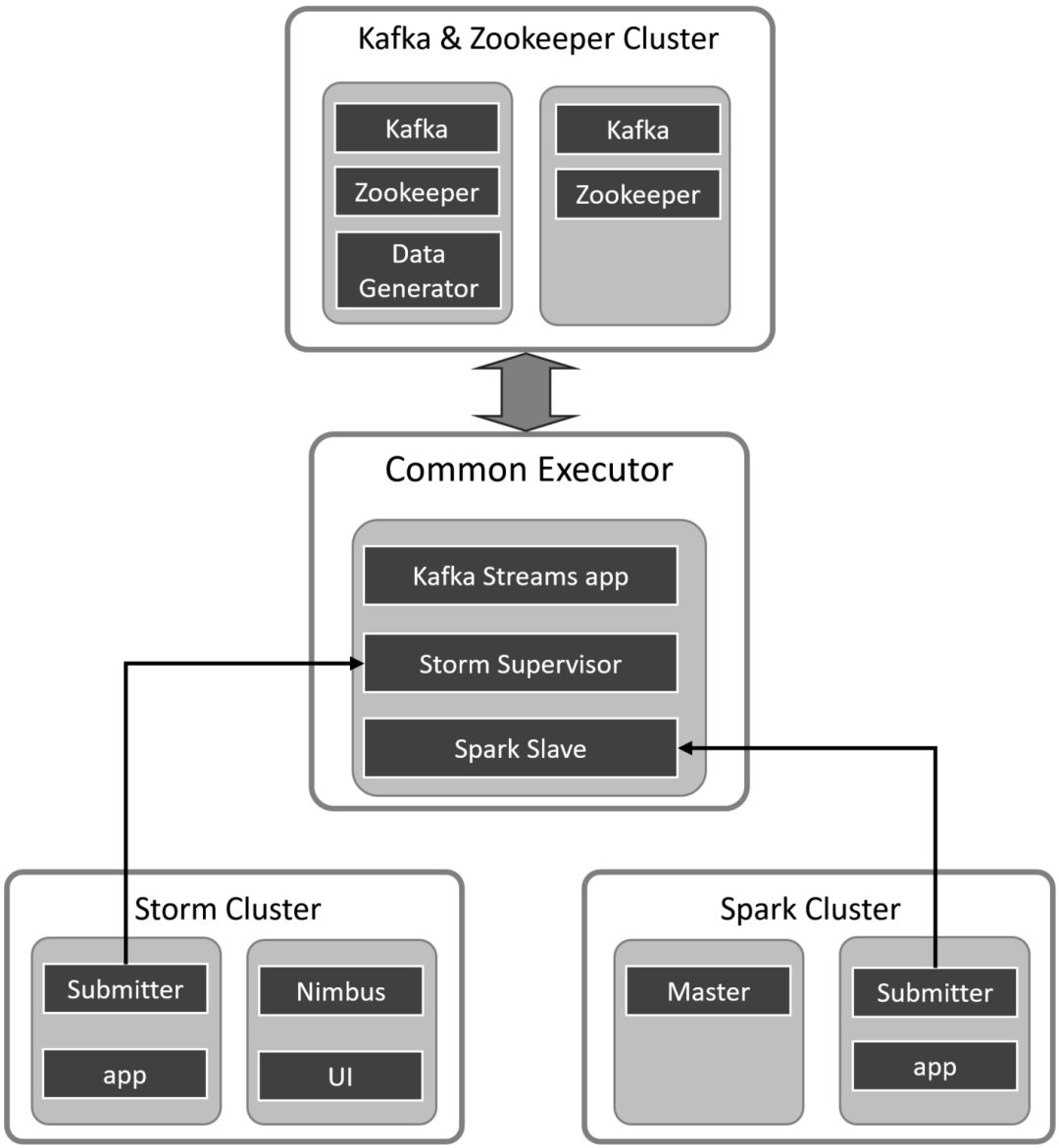

Figure 4.1 Architecture of the clusters in the experiment

The frames with the label cluster in the above figure represents an independent cluster. The Kafka and Zookeeper cluster as the common service has been introduced. The left bottom one is the Storm cluster and the right bottom one is the Spark cluster. Each cluster is configured with the same Amazon Web Services (AWS) EC2 virtual machine instances except submitter machine in Spark cluster. It has two CPU cores, because Spark requires at least two CPU cores for interaction with a human, one core for reading data and the other one for outputting information to the terminal.

On Storm cluster, there are two nodes. One of them is deployed with Nimbus and UI components. Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures. The tasks will be assigned to the machine which is the common executor. The UI component is the web tool for monitoring and managing the tasks and the cluster. The author does not want to give too much tasks to the Nimbus node to keep it stable, so the other node is used for developing the Storm application and submitting it to the cluster.

On Spark cluster, the cluster has one node as the master and one node as the submitter. Spark has independent resource management, so it does not rely on Zookeeper or have other dependencies. The master node is only responsible for resource management. The author uses the other node to develop Spark Steaming application and submit it to Spark cluster because that node has two CPU cores.

The screenshot in figure 4.2 shows the real AWS instance type in this experiment. Most of the instances are configured with AWS m3.medium virtual machines which have 1 CPU cores and 1GB RAM. Spark Submitter instance has a higher profile, t2.medium instances with 2 CPU cores and 4GB RAM. The common executor instance has the highest configuration c4.2xlarge among all the clusters. It has 8 CPU cores and 15 GB RAM. The idea is to let the applications in this experiment can execute computing tasks efficiently.

Niu-Common-Executor	i-8dd84d06	c4.2xlarge
Niu-Kafka-zk-1	i-742dc345	m3.medium
Niu-Kafka-zk-2	i-41d13e70	m3.medium
Niu-Spark-Submitter	i-6a2fd85b	t2.medium
Niu-Spark-Master	i-54961bdf	m3.medium
Niu-Spark-1	i-cab559fb	m3.medium
Niu-Spark-2	i-cbb559fa	m3.medium
Niu-Spark-3	i-efdc4964	c4.2xlarge
Niu-Storm-Nimbus	i-8136cbb0	m3.medium
Niu-Storm-Submitter	i-8232cfb3	m3.medium
Niu-Storm-2	i-3332cf02	m3.medium

Figure 4.2 AWS instance list

4.3 Program Flow in the Experiments

Besides integrating Kafka and Zookeeper into an independent cluster as the common public service, and giving the same AWS instances for running applications, the Program Flow is the next thing that guarantees the fairness in comparing the three real-time technologies in the aspect of ease of programming. All the stream processing applications are programmed to strictly follow the same program flow. Readers can find out their own subjective impressions on the difficulties to program with the three technologies. Here is the program flow:

1. Acquire data from Kafka.

2. Transform data format.
3. Group by key (Patient ID).
4. Apply algorithm.
5. Aggregation (Reduce, Join).
6. Output (Kafka, Database, HDFS, REST API)

The whole flow is defined by the author based on the data flow in Nokia Technologies. It does not cover everything needed for remote patient monitoring. In this thesis they are the basic data processing steps. Section 4.4 'Programs' will compare the minimum codes required to implement the program flow for each technology. Step 1 and 6 are just their literal meanings as the input and output of the experiment. Step 2, transforming data format, is to transform the raw data into the format so that the program can extract the key of a record in the data stream. Step 3, group by key, this step aims to divide the original data streams into multiple sub-streams each of which has a unique key (patient ID), so that each sub-stream only contains a unique patient's data. Step 4, applying algorithm, it is to demonstrate how to apply user defined algorithms to process each of the sub-stream data. Step 5, aggregation is to merge the sub-streams into new streams for output.

4.4 Comparing General Ease of Programming

In this experiment, an ECG *RPeak Detection* algorithm is applied to simulate the RPM scenario. The RPeak Detection algorithm identifies the R wave of an electrocardiogram (ECG) signal. Figure 4.3 illustrates the ECG signal and R wave. The R wave is the first upward detection of the QRS complex and is followed by a downward S wave.[24]

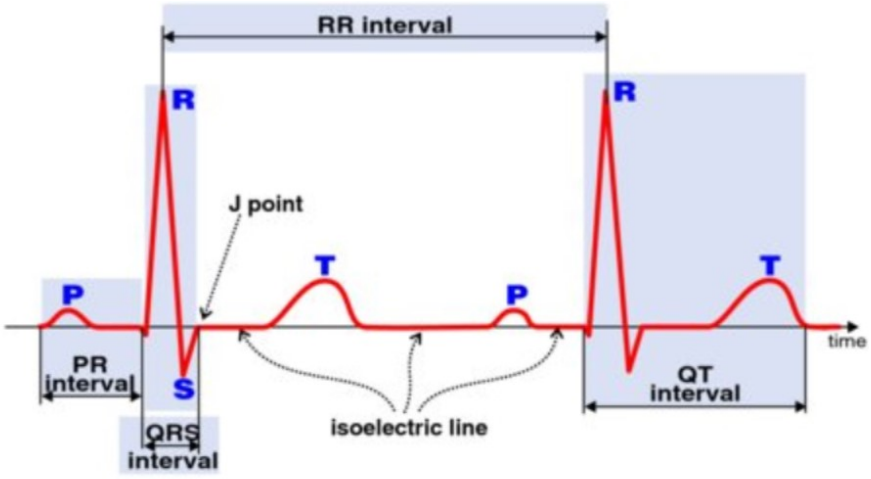

Figure 4.3 Electrocardiogram

The RPeak Detection algorithm is a Python library developed by Nokia Technologies. In this experiment, we have used it just to demonstrate how to integrate external programs. The algorithm in the experiment could be anything. RPeak Detection was selected because Nokia Technologies needs it and the author wants to demonstrate how to integrate it to the three technologies.

Since the selected three technologies have different levels of multi-language support, both Java and Python languages in implementing the experiment. Spark Streaming provides a set of full high-level Python APIs. Thus Spark Streaming application programs can be developed using only just Python. In contrast, Kafka Streams only supports Java APIs, and Storm supports most mainstream programming languages only for defining Spouts and Bolts. The main Storm topology still requires Java as the programming language. Therefore, in the text below, the source code snippets to be compared are written in Python for Spark Streaming, and in Java and Python for Storm and Kafka Streams.

4.4.1 Comparing Configuration and Topology

	Configuration and Topology

Storm (Java)	`//Configuration for Kafka Connection`
	`String topic = "streams-file-input";`
	`BrokerHosts brokerHosts = new ZkHosts("Niu-Kafka-0:2181");`
	`SpoutConfig spoutConf = new SpoutConfig( brokerHosts, topic, zkRoot, id );`
	`spoutConf.scheme = new SchemeAsMultiScheme( new StringScheme() );`
	`spoutConf.startOffsetTime = kafka.api.OffsetRequest.LatestTime();`
	
	`//Create TopologyBuilder`
	`TopologyBuilder stormBuilder = new TopologyBuilder();`

Spark Streaming (Python)	`//Configuration for Kafka Connection`
	`topic = 'streams-file-input'`
	`conf = SparkConf()`
	
	`// Create object StreamingContext`
	`sc = SparkContext(appName="ECGSparkStreaming", conf=conf)`
	`sparkStreamingContext = StreamingContext(sc, 1)`

Kafka Streams (Java)	`//Configuration for Kafka Connection`
	`String topic = "streams-file-input";`
	`Properties props = new Properties();`
	`props.put(StreamsConfig.APPLICATION_ID_CONFIG, "app-kafka-streams-mapper");`
	`props.put(StreamsConfig.ZOOKEEPER_CONNECT_CONFIG, "Niu-Kafka-1:2181");`
	`props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");`
	`props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG, Ser-des.String().getClass().getName());`
	`props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG, Ser-des.String().getClass().getName());`
	
	`//KStreamBuilder will create KStream object later`
	`KStreamBuilder kafkaStreamsbuilder = new KStreamBuilder();`
	`KafkaStreams streams = new KafkaStreams(kafkaStreamBuilder, props);`
	`streams.start();`

Table 4.4 Comparing the minimum codes for configuration and topology

Table 4.4 shows the minimum required source code to configure Kafka connection, Zookeeper connection and to construct a topology object for each technology. The amount of code and content and style are quite similar. Most of the source codes is used just for defining and initializing variables. Thus, the author gives equal scores to all the technologies in terms of ease of programming.

4.4.2 Comparing Reading Data from Kafka

	Reading data from Kafka

Storm (Java)	`//Create a Kafka Spout object to read data from Kafka`
	`KafkaSpout kafkaSpout = new KafkaSpout( spoutConf );`
	`//Add the Spout to the topology with parallelism 2`
	`stormBuilder.setSpout( "kafka-spout", kafkaSpout, 2 );`

Spark Streaming (Python)	`#Create two streams to subscribe Kafka`
	`num_streams = 2`
	`kafka_streams = [ KafkaUtils.createStream(ssc, zkQuorum, "spark-streaming-consumer", {topic: 4}) for _ in range(num_streams) ]`
	`#Combine the two KStreams into one stream object`
	`union_stream = ssc.union(*kafka_streams)`
	`#slines is the new KStream object that owns the data`
	`lines = union_stream.map(lambda x: x[1])`

Kafka Streams (Java)	`final Serde<String> strSerde = Serdes.String();`
	`final Serde<Long> longSerde = Serdes.Long();`
	
	`//Create a KStream object to read data from Kafka`
	`KStream<String, String> source = kafkaStreamsbuilder.stream( strSerde, strSerde, topic);`

Table 4.5 Comparing the minimum codes for reading data from Kafka

The minimum required source code for reading data from Kafka are all quite short for all the selected three technologies. The parallelisms of subscribing data from Kafka are all set to two, which means there are two parallel processes or threads as the consumer for each technology. The author's opinion on the difficulty of programming for the three technologies is that they are same.

4.4.3 Comparing Data Format Transformation

	Transforming data format

Storm (Java)	`// Define a Bolt class to transform data form to a new one`
	`public static class SplitBolt extends BaseBasicBolt {`
	`  @Override`
	`  public void execute( Tuple tuple, BasicOutputCollector collector ){`
	`  String patientID = "";`
	`  String data = tuple.getString(0);`
	`  try{`
	`  JSONObject obj = new JSONObject( data );`
	`  patientID = obj.getString( "p" );`
	`  JSONArray sensers = obj.getJSONArray("s");`
	`  for( int i=0; i < sensers.length(); i++ ){`
	`  JSONObject item = sensers.getJSONObject(i);`
	`  JSONObject meta = item.getJSONObject("meta");`
	`  String type = meta.getString("t");`
	`  if( type.equals( "N2-ECG-1" ) ){`
	`  data = item.toString();`
	`  break;`
	`  }`
	`  }`
	`  } catch ( Exception e ){}`
	`  //Send the patient id and its data in pair to next bolts`
	`  collector.emit(new Values( patientID, data) );`
	`  }`

	``
	    @Override
	    public void declareOutputFields(OutputFieldsDeclarer de-
	clarer){
	        declarer.declare(new Fields("patientID", "data"));
	    }
	}
	
	//Add the Bolt to the topology with parallelism 2 and config it
	//to receive data from Kafka Spout.
	stormBuilder.setBolt("split", new SplitBolt(), 2)
	            .shuffleGrouping("kafka-spout");
	``

Spark Streaming (Python)	```
	#Define a function mapper to transform data format
	def mapper(line):
	# unpack json
	    raw_dict = json.loads(line)
	    patient_id = raw_dict['p']
	    ecg_data = {}
	    for item in raw_dict['s']:
	        if item['meta']['t'] == 'N2-ECG-1':
	            ecg_data = item
	            break
	
	    output_pack = json.dumps(ecg_data)
	
	    return (patient_id, output_pack)
	
	#Call the function 'mapper' and create a new object 'streams'
	#which contains the new format of the data
	streams = lines.map(mapper)
	``

Kafka Streams (Java)	```
	//Create a new KStream object 'ECG' after transforming data
	//format
	KStream<String, String>
	ecg = source.map(new KeyValueMapper<String, String,
	    KeyValue<String,String>>() {
	      @Override
	      public KeyValue<String, String> apply(String key, String
	value){
	         String patientID = "-1";
	         String data = "";
	         try{
	            JSONObject obj = new JSONObject(value);
	            patientID = obj.getString("p");
	            JSONArray sensers = obj.getJSONArray("s");
	            for(int i=0; i < sensers.length(); i++){
	               JSONObject item = sensers.getJSONObject(i);
	               JSONObject meta = item.getJSONObject("meta");
	               String type = meta.getString("t");
	               if(type.equals("N2-ECG-1")){
	                  data = item.toString();
	                  break;
	               }
	            }
	         } catch(Exception e){}
	         return new KeyValue<String, String>(patientID, data);
	      }
	    });
	``

Table 4.6 Comparing minimum codes for transforming data format

In transforming data formats, the function of the source code is to unpack the received JSON strings and extract patient ID info and ECG data, and then package them as a pair

and send the data out to the downstream process. From the source code, it is apparent that the source code required by Spark Streaming is much shorter than the code written for the other two technologies. Spark Streaming and Kafka Streams both provide similar style of high-level API 'map' function to execute UDF(User Defined Function) on the received data. From the author's programming experience, the source code of the two technologies are easy to read. However, for Storm a low-level API is provided that requires programmers to inherit a Bolt class such as BaseBasicBolt. There are plenty of Bolt class alternatives with similar functions. The author often felt confused when choosing the right class to inherit. Compared to Storm, the higher-level APIs of Spark Streaming and Kafka Streams are quite limited and easy to choose. Therefore, the author ranks Spark Streaming and Kafka Streams easier to use than Storm in terms of code amount and ease of programming.

4.4.4 Comparing Data Aggregation and Multi-Language Support

	Aggregation & Integrating Algorithm R-Peak Detection

Storm (Java & Python)	```
	/////////////// Start of Java codes ///////////////////
	//Define a Storm ShellBolt for executing foreign language codes
	public static class ECGShellBolt extends ShellBolt implements IRichBolt {
	   public ECGShellBolt(){
	      super("python", "RPeakBolt.py");
	   }
	
	    @Override
	    public void declareOutputFields(OutputFieldsDeclarer declarer){
	       declarer.declare(new Fields("patientID", "data"));
	    }
	
	    @Override
	    public Map<String, Object> getComponentConfiguration() {
	       return null;
	    }
	}
	
	//Last, Add the Bolt to the topology and config it to receive data from the Split Bolt
	stormBuilder.setBolt("ecg_shell_bolt", new ECGShellBolt(), 2)
	       .fieldsGrouping("split", new Fields("patientID"));
	/////////////// End of Java codes ///////////////////
	
	///////////// Start of Python codes RPeakBolt.py ////////////
	//define a Python program to execute RPeak Detection algorithm
	class RPeakBolt(storm.BasicBolt):
	    def __init__(self):
	        self.patients = {}
	
	    def process(self, tuple):
	        patient_id = tuple.values[0]
	        package = json.loads(tuple.values[1])
	
	        meta = package['meta']
	        data = package['data']
	``

```
        if self.patients.has_key(patient_id ) == False:
            self.patients[patient_id] = rpeak.RPeakDetector(
fs=meta["rate"] , ecg_lead="MLI")

        algo = self.patients[patient_id]
        peaks, rri = algo.analyze( data )

        storm.emit( [patient_id, meta['n']] )

RPeakBolt().run()
//////////////  End of Python codes  ////////////////////
```

Spark Streaming (Python)

```
def reducer( line ):
# unpack json
    raw_dict = json.loads( line )
    patient_id = raw_dict[0]
    ecg_data = raw_dict[1]

    algo = rpeak.RPeakDetector( fs=ecg_data['meta']['rate'],
ecg_lead='MLI')
    peaks, rri = algo.analyze( ecg_data['data'] )
    output_data = {
        'peaks':peaks,
        'rri':rri,
    }
    output_pack = json.dumps( output_data )
    return ( patient_id, output_pack )

#Call the function 'reducer and create a new object 'streams'
#which contains the new content of the data
streams = streams.reduceByKey(reducer)
```

Kafka Streams (Java & Python)

```
//group the stream into KTable data struct by the key
KTable<String, String> kTable = ecg.reduceByKey(
    new Reducer<String>(){
        @Override
        public String apply(String V1, String V2){
            ExecPy objExec = new ExecPy();
            String strRPeak = objExec.exec( V2 );

            return strRPeak;
            }
        }, "ecg-table-reduce"
      );

////////////// Start of Rpeak.py //////////////
//Rpeak.py uses the rpeak python lib to detect the R peak from
the received data
if len(sys.argv) < 2:
      exit(-1)

package = json.loads(sys.argv[1])
meta = package['meta']
data = package['data']

algo = rpeak.RPeakDetector( fs=meta['rate'], ecg_lead="MLI" )
peaks, rri = algo.analyze( data )
peak_list = [peaks, rri]
print peak_list
////////////// End of Rpeak.py //////////////

////////////// Start of ExecPy.java //////////////
```

```
//
public class ExecPy {
    public String exec( String data ){
        JSONObject obj = new JSONObject( data );
        List<String> cmd = new ArrayList<String>();
        cmd.add("python");
        cmd.add("../../RPeak.py");
        cmd.add( data );

        ProcessBuilder pb = new ProcessBuilder( cmd );
        pb.redirectErrorStream(true);
        String line = "";
        try{
            Process process = pb.start();
            InputStream is = process.getInputStream();
            InputStreamReader isr = new InputStreamReader(is);
            final BufferedReader br = new BufferedReader(isr);
            line = br.readLine();
        }catch(IOException e){
            System.out.println("io error: "+ e);
        }

        return line;
    }
}
/////////////// End of ExecPy.java //////////////
```

Table 4.7 Comparing minimum codes for aggregation and integrating algorithm

The actual function of data aggregation in this experiment is to divide the formatted data received from the previous step into subgroups, in each of which the data have the same key: patient ID. This means that the data in each subgroup belongs to one patient so that it is practical to detect R-Peak for each patient individually.

It is obvious that Kafka Streams codes amount is bigger than the others in this section. Because it has some extra codes from ExecPy.java. This program wraps the minimum codes to execute external program written by other language other than Java, such as Python. In this experiment, ExecPy.java is called to execute Rpeak.py, so that the main Kafka Streams application program can pass data to Rpeak.py and the get back the result of RPeak detection. For comparing the code amounts of the three technologies, the author excludes the ExecPy.java from Kafka Streams. The reason is the program is a one-time use. Developers can reuse it once it is developed.

In general, the API frameworks provide functions to let programmers focus on the logical codes to process the data in the subgroups, such as Spark Streaming's 'reduceByKey' and Kafka Streams' 'reduceByKey' functions. Storm provides an API called 'fieldGrouping' to do the same work. However, Storm only promises that the data with the same key will be delivered to the same executor but it does not guarantee that the same executor will receive the data with the same key. Thus, there is a high possibility that one executor of a Storm application will receive data with multiple different keys. Programmers have to write Bolt codes to take care of this situation. In the author's opinion, Storm's design on data aggregation can bring annoying troubles and extra work to programmers. In summary, the author found Spark Streaming and Kafka Steams much more intuitive to use in

the area of data aggregation. In contrast, Storm is not very friendly to programmers in this area.

Since the original R-Peak Detection algorithm was written in Python, the author needed to compare the ease of supporting and integrating multiple programming languages. Spark Streaming has full API support for Java, Scala, Python and R. Thus, in this experiment the author only needed to write Python code to integrate R-Peak Detection. Since Kafka Streams only have Java API, the author wrote a Python program 'RPeak.py' to integrate R-Peak Detection library and a Java program 'ExecPy.java' to execute external program 'RPeak.py'. The R-Peak Detection integration source code is quite similar to the code for Spark Streaming. Thus, the author thinks the difficulty of multi-language support of Kafka Streams is only a little bit higher than Spark Streaming. Storm provides 'basicBolt' class to inherit for every language it supports. Defining a new class inherited from 'basicBolt' and integrate R-Peak Detection library is not difficult. The only annoying thing is that programmers have to take care of the multi-keys situation. In summary, the author found the Spark Streaming programming model as the easiest and Storm programming the hardest in this area, even though Storm supports a wider range of languages.

4.5 Time cost

Performance is yet another critical factor when choosing technologies. Although the thesis emphasizes on the architecture, ease of programming and design philosophy, it would be necessary to test the performance of the three technologies. After all, a poor performance is not practical in applying a technology in real production environment, no matter how well the technology is designed. Considering the three technologies are chosen for analyzing data in real-time, time cost is definitely one of the critical factor to measure the performance of a technology. Thus this chapter compares the time cost of each technology.

4.5.1 Method

Chapter 4.3 introduces the program flow in the experiment. The flow defines the steps that programs should follow. It starts from reading data from Kafka, then transform data, group by key, apply algorithm, aggregation and finally end with output data. The figure 4.8 illustrates the time consumption to be measured in the program flow. The first step and the last step do not involve any computing tasks, so the latency test skips these two steps and measures the time consumptions of the steps between them. Especially the step transformation and applying algorithm, they usually have intensive CPU consumption and could have impact on the overall latency. Of course, transferring data between these steps may also cause latency. Therefore, time consumption of step transformation and step applying algorithm and the total time consumption will be measured and compared for the three technologies.

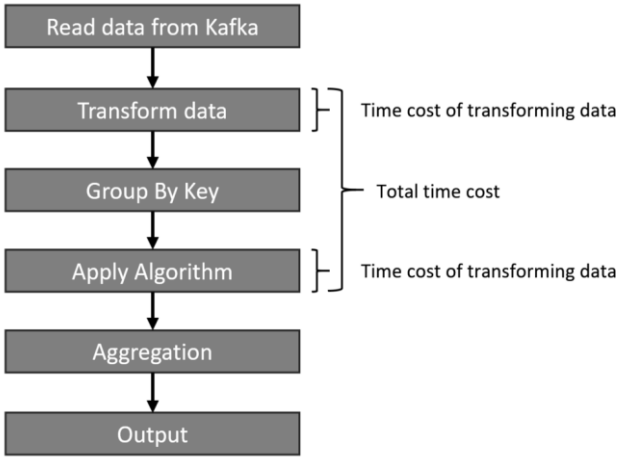

Figure 4.8 Time consumption measurement

4.5.2 Latency of Storm

In the experiment, 811 records are processed by the Storm application. Figure 4.9 shows the distribution of the time cost of transformation step. The X axis is the record ID and the Y axis is time cost for each record in milliseconds. It's apparent that Storm performs quite stable and high efficient in transforming data.

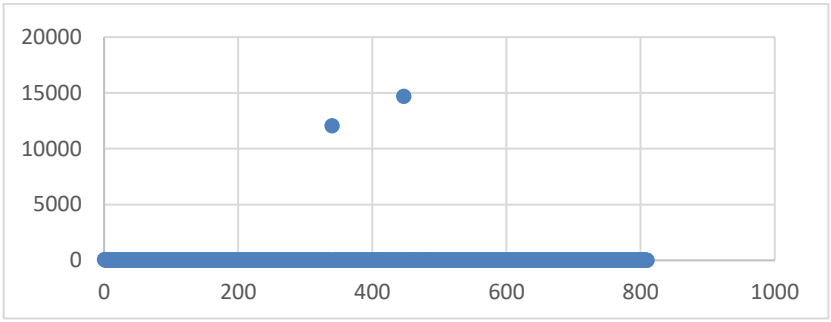

Figure 4.9 Transformation time cost of Storm application

Next, figure 4.10 shows the time cost of applying the algorithm RPeak Detection. In most situation, Storm performs stably, the time cost stays in range from 20ms to 35ms. But it is also obvious that records from 200 to 400 take some more time.

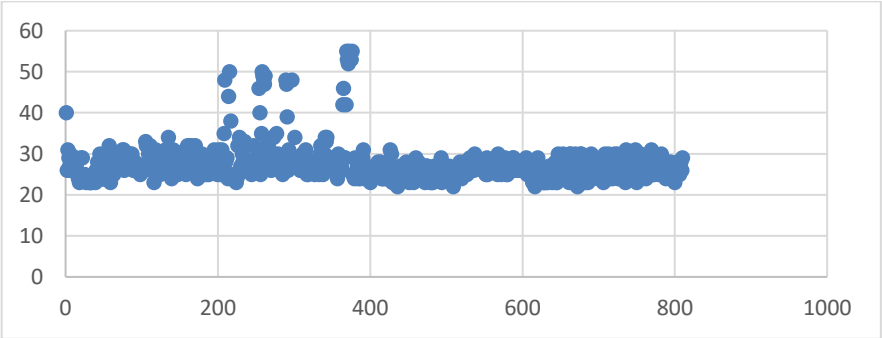

Figure 4.10 Applying algorithm time cost of Storm application

Last, figure 4.11 shows the total time cost distribution. The conclusion is obvious that the time cost grows strongly and unstably. Start from record ID 260 approximately, it takes at least 1.5 seconds to finishing processing a record. This may be slightly caused by the step applying algorithm as the figure 4.10 shows. But network, scheduler may also cause the huge latency in processing data.

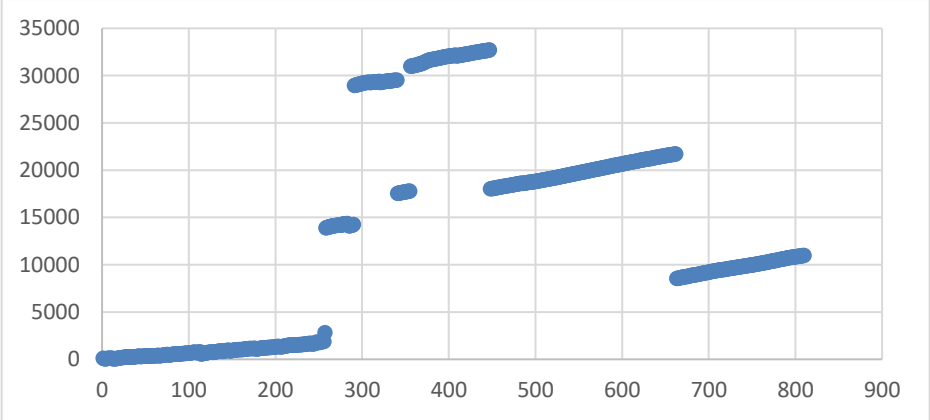

Figure 4.11 Total time cost of Storm application

4.5.3 Latency of Spark Streaming

In the experiment, 1560 records are collected to measure the latency of Spark Streaming. Figure 4.12 shows that the time cost is very stable and low, 4 to 6ms. In the step transformation, Spark Streaming basically has the same good performance as Storm.

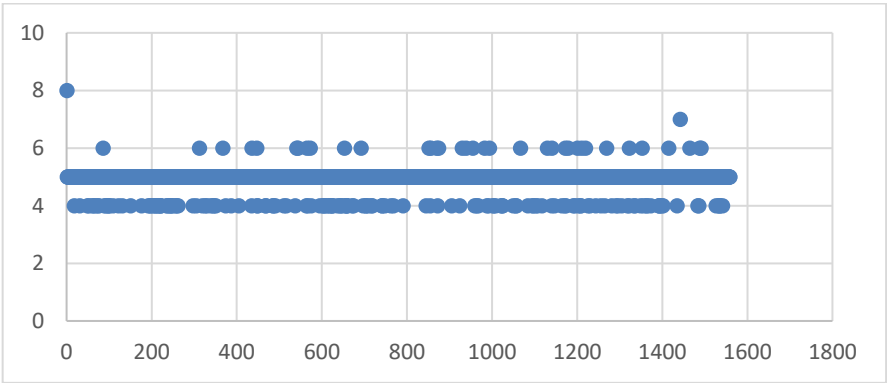

Figure 4.12 Transformation time cost of Spark Streaming

Next is the time cost of applying algorithm RPeak Detection in Spark Steaming application. Figure 4.13 shows that Spark Streaming performs quite stable and efficient. Most time cost keep in the range from about 17 to 27ms.

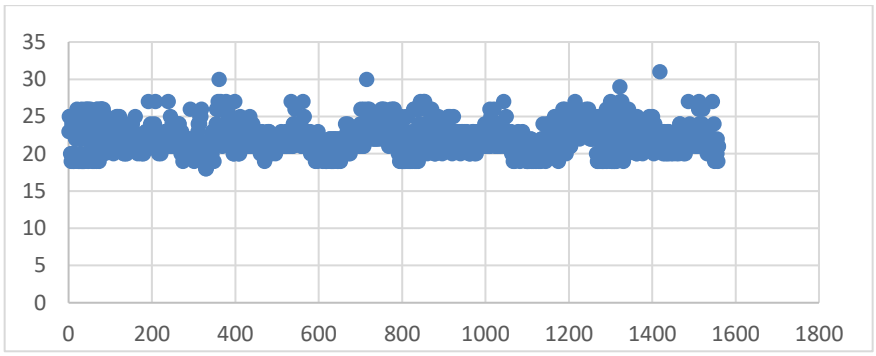

Figure 4.13 Applying algorithm time cost of Spark Streaming

Last, it is the total time cost measurement. As the figure 4.14 shows, Spark Streaming performs very steadily. But the total time cost grows gradually to around 2.3 seconds. This may not be a desired result in real practice.

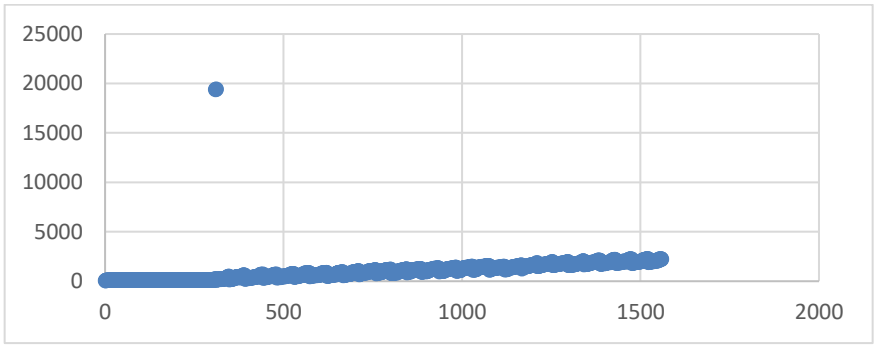

Figure 4.14 Total time cost of Spark Streaming

4.5.4 Latency of Kafka Streams

In the experiment, 44359 messages are recorded to test the latency of Kafka Streams. Figure 4.15 shows that Kafka Stream has an extremely static and low time cost, less than 30ms.

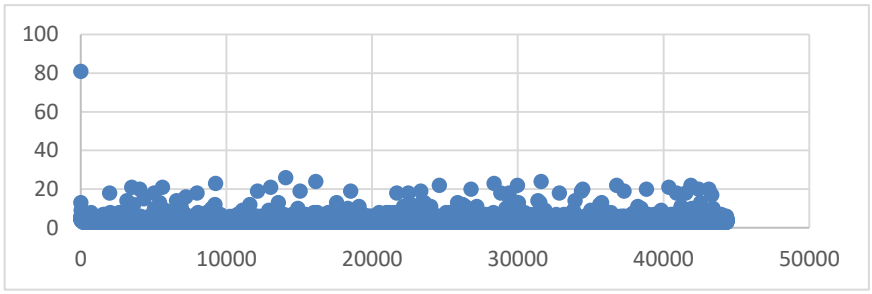

Figure 4.15 Transformation time cost of Kafka Streams

The distribution of time cost of applying algorithm is a little bit dynamic, but overall, the time costs stay lower than 30ms as figure 4.16 shows. This is quite a good performance.

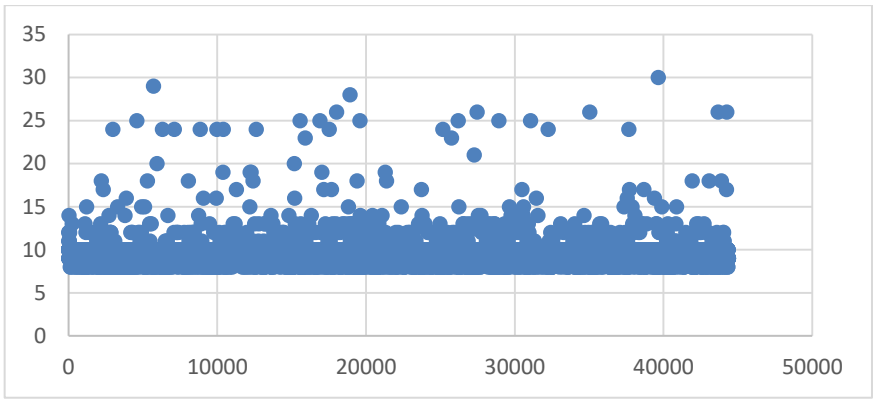

Figure 4.16 Applying algorithm time cost of Kafka Streams

And Overall, the total time costs keep smaller than 40ms from beginning to the end. This latency is much lower than Storm and Spark Streaming.

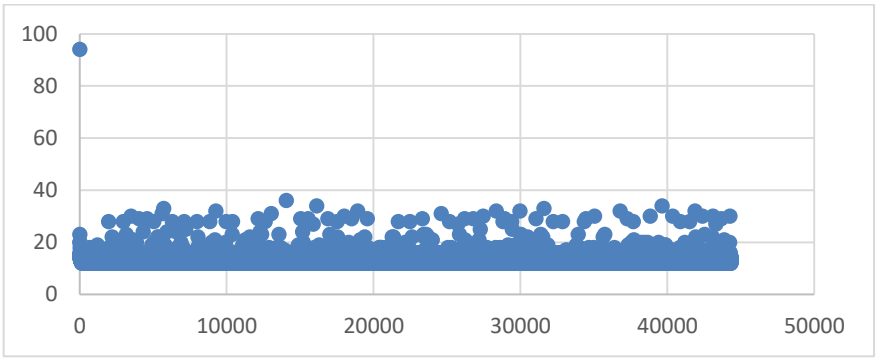

Figure 4.17 Total time cost of Kafka Streams

In this experiment, all the three applications are configured with only one parallelism. That means Storm application is configured with one worker process to execute computing tasks. And Spark Streaming application is configured with one receiver and one DStream object. Kafka Streams application is launched with one Java process. This result may not represent the performance when they run in cluster mode. In the real world, all the clusters and applications will be optimized. Their official websites all provide the methods to tune the applications. Therefore, the author believes that understanding the insights of each technology is more important. Next chapter, the thesis compares the design philosophies inside every technology.

5. COMPARING THE DESIGN PHILOSOPHY

Apache Storm, Spark Streaming and Kafka Streams are all excellent stream processing technologies. After years of development and evolution, they are quite similar to each other in terms of their key features: distribution, scalability, low-latency, high-availability, fault-tolerance and guaranteed message processing. However, they were originally created for solving different problems. This chapter goes into the technology developers' minds to compare the design thinking behind the systems and find out the differences.

5.1 Comparing Background and Motivation

5.1.1 Motivation of Storm

Storm was born for real-time analytics. The author of Storm, Nathan Marz, records the history of Storm in his personal blog. Initially, Storm was born in a company called BackType. BackType built analytics products to help businesses understand their impact on social media both historically and in real-time. Before Storm, the real-time portions of the implementations were done using the standard queues and workers approach. For example, they write the Twitter firehose to a set of queues, and then Python workers would read those tweets and process them. Oftentimes these workers would send messages through another set of queues to another set of workers for further processing. But the system was brittle, and they have to make sure the queues and workers all stayed up. It is cumbersome to build apps, since most of the logic have to do with sending/receiving messages, serializing/deserializing messages and so on.[25] The fact that all that logic is self-contained in one application did not seem right to Nathan.

5.1.2 Motivation of Spark Streaming

Spark was originally targeted at interactive queries and iterative computation. An interview with UC Berkeley Professor and Databricks CEO Ion Stoica recalled the early days of Spark. Everything started from a class project, Mesos, that he taught in the spring of 2009. To demonstrate that it was actually easier to build a new framework from scratch on top of Mesos, and to target a field in which Hadoop was not good enough, Ion and his students chose interactive queries and iterative computation, such as Machine Learning.[26]

5.1.3 Motivation of Kafka Streams

The goal of Kafka Streams is to simplify stream processing enough to make it accessible as a mainstream application programming model for asynchronous services[27]. In the official introductory blog of Confluent.io (a startup company founded by the authors of Apache Kafka), the authors noted that there is wealth of work happening in the stream

processing area, e.g., Apache open source framework Spark, Storm, Flink and Samza and proprietary services such as Google's Data Flow and AWS Lambda. The gap that Kafka Streams aims at filling is less in the analytics domain but more in building core applications and microservices that process data streams. So Kafka Streams, a component of Apache Kafka, is a library for building highly scalable, fault-tolerant, distributed stream processing applications.

5.1.4 Summary

This thesis compares Storm, Spark Streaming and Kafka Streams in the context of real-time analytics. After years of development, now their feature sets have quite a big overlap, such as they are all open source, distributed, real-time and so on. However, as introduced above, their original backgrounds and motivations are quite different from each other. That caused the differences in their internal designs. This is also one reason that stops engineers migrating from one technology to another. This chapter compares the design philosophy and implementation of the three systems in a number of areas, beginning from their data structures.

5.2 Comparing Data Structures

A common core abstraction of Storm, Spark Streaming and Kafka Streams is the *stream* concept. The stream data structure design affects the whole architecture and it is also the key to understanding each of them.

5.2.1 Data Structures of Storm

A stream in Storm is a distributed abstraction. A stream represents an abstracted continuous data flow. Streams are produced and processed in parallel. As figure 3.2 illustrates, a spout produces brand new streams, and a bolt takes in streams as input and produces streams as output.

Inside the stream, there is the data structure 'tuple' which is introduced in chapter 3.1.2. A stream is an unbounded sequence of tuples. The inputs and outputs of spouts and bolts are actual tuples, in other words, it is the tuples that actually pass through the whole application. A tuple is a named list of values, and a field in a tuple can be an object of any type. That is to say, the tuple is the data structure that stores the real data that a Storm application processes.

5.2.2 Data Structures of Spark Streaming

Similar to Storm, Spark Streaming also defines an abstraction, DStream (Discretized Stream) that represents a continuous stream of data. But different from Storm, it would be inappropriate to consider DStream as a parallel stream, because internally a DStream is a continuous series of RDDs. All the operations on a DStream will be translated to

operations on an RDDs. And the operations on RDDs will be computed by the Spark Engine.

In general, one of the most important data structures of Spark is the RDD. An RDD (resilient distributed dataset) is a partitioned, distributed collection of elements that can be operated on in parallel. Different from Storm's tuples, which pass through a Storm application as the inputs and outputs of the parallel executor threads, RDDs are immutable once sliced and distributed over a set of machine's memories. An RDD defines two types of operations: transformation and action. Transformations are lazy operations that only define a new RDD, while actions launch a computation to return a value to the program or write data to external storage.

5.2.3 Data Structures of Kafka Streams

Kafka Streams does not introduce any new data structures for stream processing. Kafka Streams simplifies application development by building on the Kafka producer and consumer libraries, and leverages the native capabilities of the underlying Kafka system to offer data parallelism, distributed coordination, fault tolerance, and operational simplicity.

Inside Kafka, there is a distributed publish/subscribe message queue data structure. What is very different from Storm and Spark that use in-memory data structures, the queues of Kafka actually store data on disks. There is a general perception that "disks are slow", but in fact, the performance of disk can be much faster than people expect. As a result, the performance of linear writes on a JBOD (Just a Bunch of Disks/Drives) configuration with six 7200rpm SATA RAID-5 array is about 600MB/sec. JBOD Is an architecture using multiple hard drives exposed as individual devices.[28] But the performance of random writes is only about 100k/sec—a difference of over 6000X. These linear read and write operations are heavily optimized by the modern operating system[29].

5.2.4 Summary

Table 5.1 gives a simple summary of the core data structures of each technology. Storm, Spark Streaming and Kafka Streams have the same concept 'stream'. But their data storage unit and data location are different.

	Storm	Spark Streaming	Kafka Streams
Stream	A continuous unbounded data flow	A continuous unbounded data flow	A continuous unbounded data flow
Data storage unit	Tuple: a named list of values	RDD: a read-only distributed dataset with operations	Queue
Data location	In-memory	In-memory	In-disk

Table 5.1 Comparison of data structures

5.3 Comparing Parallelism & Scalability Mechanisms

Because of Apache Hadoop's popularity in Big Data world, scalability has become a compulsory feature for the design and implementation of data analytics technologies. Since data sizes can grow fast and overcome a single server system's capacity quickly, the ease of scaling up a cluster is also an important aspect for consideration. There will not be much time for engineers to expand the capacity of a cluster before overflow. Making the data storage distributed or making the data computing parallel are the main two methods to achieve scalability for a framework. This section compares the scalability strategy for each technology.

5.3.1 Parallelism & Scalability in Storm

Different parts of the Storm topology can be scaled individually by tweaking their parallelism. This feature makes Storm elastic to customize the parallelism of a topology. A Storm cluster can run multiple Storm applications at the same time. To scale up a Storm application's throughput has two possible scenarios. If the cluster has extra resources then just tune the application, otherwise scale up both.

Scaling up or down is easy for Storm applications, but it is necessary to understand the parallelism of a Storm topology before learning the methods of scaling. There are three main entities that make up the parallelism of Storm: Worker Process, Executor and Task. Figure 5.2 shows their relationship. From outside to inside, a server in a Storm cluster runs multiple processes. They are called worker processes. Each process spawns multiple threads called executors to process for a specific subset of a topology. And each executor(thread) runs one or multiple tasks. The tasks have to be the same component, either Spout or Bolt. The tasks execute the actual code that programmers write for processing data.

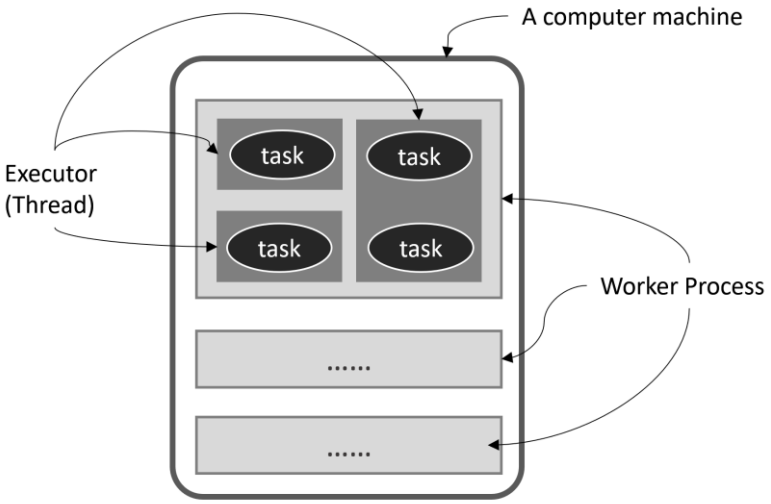

Figure 5.2 Relationship among worker process, executor and task

Scaling up or down a Storm application can be performed by adjusting not only the number of worker processes but also the number of executors and the number of tasks. Initially,

all of them can be configured in the source code. Table 5.3 is an example of a Storm application which sets up two worker processes, Spouts ‘blue’ with two executors, Bolts ‘green’ two executors and four tasks to receive data from the two Spouts executors and ‘yellow’ with six executors to receive data from the Bolts ‘blue’. In sum, the number of the parallelism for this application is to combine all the executors, 2+2+6, that is 10. There are two worker processes, so each of them shares half of the total parallelism, that is 5. Thus, each worker process will spawn 5 threads to execute the topology.

```
Config conf = new Config();
conf.setNumWorkers(2); // use two worker processes

topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2); // 2 executors
topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)   // 2 executors
               .setNumTasks(4)                              // to execute 4 tasks
               .shuffleGrouping("blue-spout");
topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6) // 2 executors
               .shuffleGrouping("green-bolt");
```

Table 5.3 An example to configure Storm’s parallelism

Figure 5.4 visually shows how these executors are divided evenly across the two worker processes. The blue Spouts emit data to the green Bolts. The green Bolts process them then transfer them to the yellow Bolts. In the example, only Bolt ‘green’ is configured with two executors and 4 tasks. Other components are not configured with tasks but just executors. So each executor for Bolt ‘green’ will share the tasks which mean each of them executes 4 / 2 = 2 tasks.

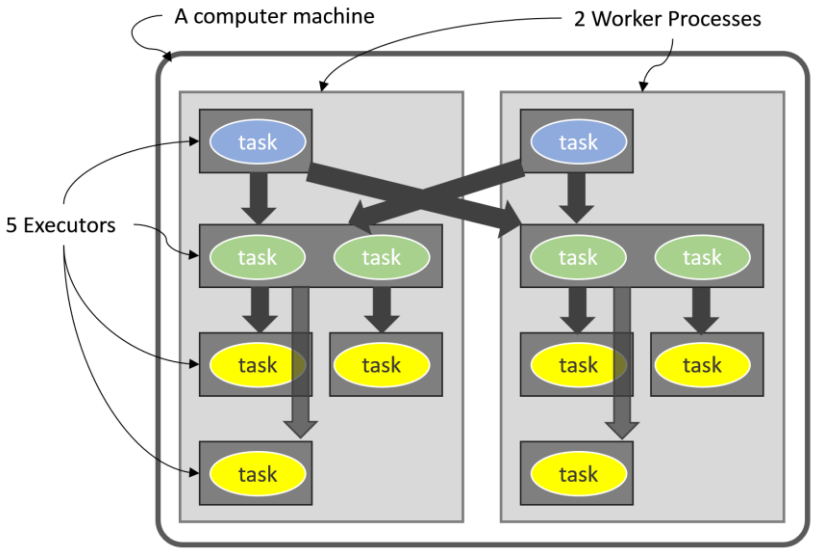

Figure 5.4 Divide tasks to executors

Besides initializing the number of worker processes, the number of executors and the number of tasks in the source codes of a Storm application, Storm also provides two options to rebalance the running status applications without being required to restart the cluster or the applications.

1. Use the Storm Web UI tool to rebalance the applications
2. Use the CLI tool to rebalance the applications

5.3.2 Parallelism & Scalability in Spark Streaming

When receiving data becomes a bottleneck in a Spark Streaming application, a natural consideration is to scale the parallelism level of receiving. By default, one DStream object creates a single receiver that receives a single stream of data. But Spark Streaming supports creating multiple input DStreams and configuring them to receive different input partitions of input data streams. The source codes in table 5.5 give an example of creating multiple DStreams:

```
numStreams = 5

kafkaStreams = [KafkaUtils.createStream(...) for _ in range (numStreams)]

unifiedStream = streamingContext.union(*kafkaStreams)
```

Table 5.5 An example to configure Spark Streaming's parallelism

Besides parallelizing the receiving data, the last thing we could do for scaling is RDD. As introduced in the previous chapter data structure, RDD is a distributed dataset that is stored in a distributed fashion and operated across the work machines. Spark tries to be as close to data source without wasting time on transferring data across network.

5.3.3 Parallelism & Scalability in Kafka Streams

Kafka Streams has close links with Kafka in the aspect of parallelism. Kafka Streams uses the core concepts of 'stream partitions' and 'stream tasks' as logical units of its parallelism. A stream partition is a sequence of Kafka messages that maps to exactly one Kafka topic partition. Considering one Kafka Streams application can subscribe messages from multiple Kafka topics, so the number of stream partitions is the sum of partitions of all the input topics.

Stream task is a fixed unit of parallelism of an application. Kafka Streams scales an application by dividing it into multiple parallel stream tasks. Kafka Streams creates a fixed number of stream tasks and assigns a list of stream partitions to each stream task. Each stream task then processes the messages received from its assigned stream partitions.

The assignment of stream partitions to stream tasks never changes once the units are created. So the number of stream tasks is fixed. Hence, the maximum parallelism of an application is the maximum number of stream tasks, which itself is the maximum number

of the input Kafka topic partitions. For example, given topic A has 3 partitions and topic B has 4 partitions, the number of the stream partitions should be sum(3, 4) = 7 and the number of stream tasks should be max(3, 4) = 4. So Kafka Streams will evenly distribute the seven stream partitions across the four stream tasks.

After understanding the parallelism of Kafka Streams, it will be easy to scale an application. Unlike Storm or Spark, Kafka Streams does not have a resource manager to automatically allocate computing resources for applications. Kafka Streams is a library that can be deployed anywhere manually. Kafka Streams also supports threading model. So there are mainly three methods for scaling.

1. Launch multiple instances of one application on one or more machines. Each instance will be assigned at least one stream task until all the tasks are assigned evenly.
2. Configure multiple threads for one instance on one machine. Each thread executes one or more stream tasks.
3. Mix methods 1 and 2.

Given that stream task is the unit of parallelism of Kafka Streams, when there is a need to scale an application, we just launch a new instance of the application. Kafka Streams will automatically re-assign one or more stream tasks to the new instances to reach a new load balance.

5.4 Comparing Fault Tolerance & Guaranteed Message Processing

Message processing strategy is another complex but critical consideration especially in terms of processing large-scale real-time data streams. Once messages are processed automatically in an unexpected way, e.g., repeated processing some messages or lost messages, that is going to lead to unpredictable results, which means the whole data processing work is performed in vain. Thus, it is compulsory for programmers to understand the mechanism of the message processing strategies for each technology before using any of them. Storm, Spark Streaming and Kafka Streams all provide three levels of guaranteed message processing: at most once, at least once and exactly once.

- At most once: every message will be processed either once or never.
- At least once: every message will be processed once or more repeatedly.
- Exactly once: every message will be definitely processed for one time only.

This chapter compares the strategies of guaranteed message processing for the three technologies.

5.4.1 Fault Tolerance in Storm

Storm's native API only supports 'at most once' (named best effort in Storm) and 'at least once' in terms of guaranteed message processing. For 'exactly once', it requires Storm's high-level API Trident that processes data streams as small batches of tuples.

Inside Storm, there is a tree data structure to track the status of each tuple and its children tuples, because a tuple from a Spout can generate many tuples based on itself. Figure 5.6 is a word count example to illustrates the tuple tree. This example calculates the count of each word in sentence 'the cow jumped over the moon'.

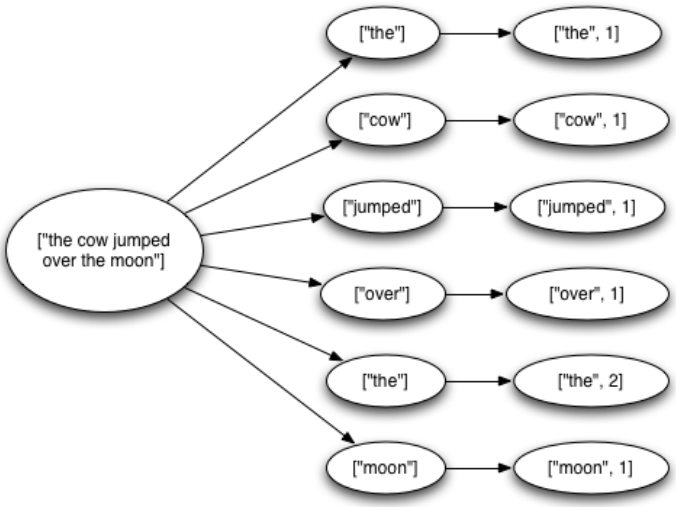

Figure 5.6 Process of wordcount with Storm[30]

```
builder.setSpout("sentences", new KestrelSpout());
builder.setBolt("split", new SplitSentence(), 10).shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 20).fieldsGrouping("split", new Fields("word"));
```

Table 5.7 Example codes of word count program with Storm

Table 5.7 shows the codes of the word count example in figure 5.6. The whole process is consist of three steps:

1. The sentence "the cow jumped over the moon" comes from a Spout and flows to a Blot 'SplitSentence'.
2. Bolt 'SplitSentence' splits it into words and emits new tuples containing the words to a new Bolt 'WordCount'.
3. Bolt 'WordCount' calculates the sum of each word from the input tuples then creates new tuples each of which contains a pair of values, word and its sum.

With the tuple tree, Storm can provide the guarantee for message processing. All the tuples get processed while passing through the Bolts. When some errors happen, Storm will replay the tuples from the Spout at the root of the tree for downstream Bolts to re-compute

the tuples. But to benefit from this guaranteed feature of Storm, programmers have to use anchoring and ack (acknowledge) or fail functions in their topology codes. Anchoring is to specify a link in the tuple tree, and it is done at the same time of emitting a new tuple. The next and last thing to do is using ack or fail function to notify Storm when the tuples processing is completed. Anchoring, ack and fail all communicate with a task named *Acker* which traces the DAG of a tuple tree from a Spout. When Acker sees that the DAG of a tuple tree is completed, failed or timeout, it sends messages to the Spout task which generates the tuples of the tree, so that the Spout task will know if it needs to replay the tuples.

So far, it is clear that the Spout task, topology's anchoring, ack and fail functions, and Acker tasks are the three compulsory conditions that guarantee the message processing. The failure of any one of them can lead to the loss of message. Thus, to achieve 'at most once' programmers should break at least one of the three conditions. To achieve 'at least once' programmers should meet every one of the three conditions. To achieve 'exactly once' programmers should use the Trident API rather than the native API.

5.4.2 Fault Tolerance in Spark Streaming

Unlike Storm, guaranteed message processing is programmed in the heart of Spark. Spark does not require any programming work for developers, which means Spark itself takes care of everything so that developers can concentrate on the business logic. And since Spark Streaming is just a stream processing library based on the core computing engine of Spark, Spark Streaming, of course, inherits the advantages of Spark core. And internally, it is RDD, the fundamental data structure of Spark core, that provides the core ability to guarantee message processing.

RDD is re-computable and distributed. Each RDD records the lineage of deterministic operations which could be replayed to re-compute the new RDDs from the data source by another executor in the cluster when an error happens to a partition of RDD. And since RDD is distributed, this mechanism can be executed in parallel. DStream is a series of RDDs and all the operations on DStream will be transformed to RDDs. So RDDs make Spark Streaming able to recover from failures.

This strategy is durable when the data sources are fault-tolerant, such as HDFS, Kafka. Spark can re-acquire data from them. But if input data of Spark Streaming applications is disposable, for example, the messages from the network will be gone forever once lost. Spark Streaming replicates the received data among multiple Spark executors in the cluster. The reliable replications are used for re-computation when errors happen to the cluster. However, this still causes two possible failure situations.

- Data received and replicated – data can be recovered if a copy of it exists on one of the other nodes.
- Data received and buffered but not replicated – the buffered data is still lost. The only way to recover this data is to get it again from the source.

Above is the failure of a worker node on which a receiver is running. If the driver node fails, then, of course, the Spark Context is lost, so all the in-memory data on the executors are lost. So far there are still risks that the received data would be lost. This is the default at most once level of guaranteed message processing.

In general, all the streaming processing applications consist of three steps: Receiving data, Transforming data and Pushing out data. To achieve the end-to-end 'exactly once' guaranteed message processing, each step must guarantee the message to be processed for once exactly. Pushing out data by default provides at least once guarantee because it depends more on the output system. So Spark only needs to guarantee the first two steps.

Now it is obvious that both receivers and driver node can fail the receiving data step, thus from version 1.2, Spark introduces a new feature 'write ahead logs' which saves the past received data to a fault-tolerant storage. With 'write ahead logs' configuration enabled, and RDD's strong fault-tolerance that guarantees to transform data successfully, Spark Streaming can provide the at least once guaranteed message processing.

In summary, Spark Streaming's guaranteed message processing depends heavily on the reliability of the data source. By default, Spark Streaming can reach 'at most once' level. With fault-tolerant data source and 'write ahead logs' configuration enabled, Spark Streaming can reach to 'at least once' level. Base on that, to reach 'exactly once' level, Spark Streaming requires the data source can provide the 'exactly once' consumption, for example, Kafka.

5.4.3 Fault Tolerance in Kafka Streams

Right now, Kafka Streams is still very young. By default, Kafka Streams only supports 'at least once' guaranteed message processing. This means Kafka Streams library guarantees that if the stream processing application fails, no data will be lost. But some data records will be consumed and processed more than once.

5.4.4 Summary

Table 5.8 gives a simple summary of the guaranteed message support of each technology.

	Storm	Spark Streaming	Kafka Streams
At most once	Supported by default	Supported by default	No
At least once	Supported by default, but requires user programming	Supported by default	Supported by default
Exactly once	Trident API	Dependent on data source	No

Table 5.8 Guaranteed message processing

6. CONCLUSION AND FUTHER DEVELOPMENT

6.1 Conclusion

The thesis compares Storm, Spark Streaming and Kafka Streams for real-time remote patient monitoring on the subject of architecture, ease of programming and design philosophy. Overall, there is no absolute best one after comparison. They are all top-level projects of Apache Foundation. In terms of features, they share a quite big range of common functions, such as distribution, real-time computation, fault tolerance, big data support and so on. From a developer's view, they all meet the requirements of developing real-time applications for processing a large volumes of data. Inside, every one of them has its own unique characteristics. The thesis presents their characteristics by comparing their architecture, ease of programming and design philosophy.

The architectures of the three technologies are compared first. It introduces the core concepts and architectures. They have a common concept 'stream' which represents abstract data flows. The rest concepts of each technology are unique. With the knowledge of the core concepts, it is easier to understand the architectures. From the thesis author's point of view, every architecture has its unique beauties and flaws. It is pointless to find out which one is better than the other ones. Understanding the architectures can help developers write correct and efficient codes while using the technologies.

The ease of programming is another important aspect but often ignored by software developers. It is important because if it is easy for programmers to develop applications, the software development cycle can be shortened so that the users can receive new software updates faster. The ease of programming is a subjective judgement. Thus the thesis introduces an experiment in which the author programs for a common target with each technology. By reviewing the programming style and code amount, readers can have an impression to know if it is easy for them. The author thinks Kafka Streams is the easiest. Because its APIs are simple and Kafka's publish/subscribe pattern is easy to understand. The second in rank is Spark Streaming. Its APIs are as simple as Kafka Streams to use. But its RDD concept takes the author a long time to understand. The last one is Storm. The author feels confused to choose APIs and give proper values to the APIs' parameters.

The design philosophy studies the genes of the technologies. Granted, the technologies in future will be much advanced than those available now. But the author believes that new technologies are developed based on the knowledge and experience learnt from existing technologies. The well-designed components could be used in other technologies. That is the reason why the author is so much interested in the design philosophies of Storm, Spark Streaming and Kafka Streams.

During the process of the thesis project, the author learned the background, motivation and design philosophy behind these technologies. These help the author get an easier start

with other similar technologies and get some ideas to create a different technology. This is the knowledge that the author wants to share with the readers. Hopefully, the information presented in this thesis can provide some help and guidance in the technical world of real-time analytics and stream processing.

6.2 Further development

With the ability to analyse human health data, new technologies are able to not only monitor patient health but also study the reasons of illnesses so that people can avoid them and stay healthy. Although this thesis focuses on comparing real-time data analytics technologies for remote patient monitoring, the analytics capabilities could have a wider range of applications, such as self-diagnosis, behaviour analysis and risk prediction. In the future by cooperating with more advanced and sensitive sensors, it could be imagined that new data analytics technologies can automatically diagnose a disease at an early stage when a person's biological data starts to change unexpectedly. Then there will be a bigger chance to cure the disease.

Network communication and scheduling are two fundamental technologies for parallel computing systems. As introduced in the first chapter, the core computing model of real-time data analytics technologies for processing huge amount of data is parallel computing. Data transfer through the network and job scheduling period have a critical effect on the overall performance of real-time analytics technologies. Spark chooses to perform all the computing work locally in order to avoid network data transfer. Kafka Streams optimizes the network communication efficiency by using OS kernel-level APIs to maximize network throughput. But Storm has to transfer data among Spouts and Bolts which might slow down its efficiency. The minimum scheduled period in Spark Streaming is one second. In contrast, Storm and Kafka Streams have no limitations on job scheduling. This limits Spark Streaming to near real-time analytics rather that hard real-time which is admittedly a pity. However, the presented technologies are developing fast all the time. The author believes that the knowledge and experience learned from these existing technologies will definitely help create even more advanced technologies, leading to even better care and patient safety.

REFERENCES

[1] Gerhard Spekowius, 2006. Advances in Healthcare Technology: 6 (Philips Re-search Book Series (closed)). 1 Edition. Springer Netherlands.

[2] Spekowius, G., 2006. Advances in Healthcare Technology. Springer Science & Business Media.

[3] R. Kohavi, N. J. Rothleder, and E. Simoudis. Emerging trends in business analytics. Communications of the ACM, 45(8):345–48, 2002.

[4] S. Tyagi. Using data analytics for greater profits. Journal of Business Strategy, 24(3):12–14, 2003.

[5] Jeffrey Dean , Sanjay Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM, v.51 n.1, January 2008 [doi>10.1145/1327452.1327492]

[6] C., H, 2013. Fundamentals of Stream Processing. 1. Cambridge University Press.

[7] A., P, 2011. Real-Time Systems Design and Analysis: Tools for the Practitioner. 4. Wiley-IEEE Press

[8] Giorgio C Buttazzo, 2011. Hard Real-Time Computing Systems: 24 (Real-Time Systems Series). 3 Edition. Springer US.

[9] Ellis, B, 2014. Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data. 1. Wiley

[10] S., G, 1994. Highly Parallel Computing (The Benjamin/Cummings Series in Computer Science and Engineering). 2 Sub. Addison Wesley Longman

[11] S., A, 2006. Distributed Systems: Principles and Paradigms (2nd Edition). 2. Pearson

[12] Jeffrey Dean , Sanjay Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM, v.51 n.1, January 2008 [doi>10.1145/1327452.1327492]

[13] http://storm.apache.org/

[14] http://zookeeper.apache.org/

[15] https://spark.apache.org/streaming/

[16] http://spark.apache.org/

[17] https://kafka.apache.org/0101/documentation/streams

[18] https://docs.confluent.io/current/streams/developer-guide/dsl-api.html

[19] https://kafka.apache.org/0110/documentation/streams/core-concepts

[20] https://flink.apache.org/index.html

[21] https://flink.apache.org/introduction.html

[22] http://stratosphere.eu/

[23] https://about.linkedin.com/

[24] X., R., 2016. ECG from Basics to Essentials. John Wiley & Sons.

[25] http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html

[26] https://www.oreilly.com/ideas/apache-sparks-journey-from-academia-to-industry

[27] https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/

[28] Rouse, Margaret (September 2005). "JBOD (just a bunch of disks or just a bunch of drives)". SearchStorage.TechTarget.com. TechTarget. Retrieved 2013-10-31.

[29] https://kafka.apache.org/081/documentation.html

[30] https://storm.apache.org/releases/1.0.6/Guaranteeing-message-processing.html

APPENDIX A: USING TEXT STYLES IN MS WORD